# Structured Parallel Programming with Trees
# (木を用いた構造化並列プログラミング)

A dissertation

by

Sato, Shigeyuki
(佐藤 重幸)

Submitted to the

Department of Communication Engineering and Informatics

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

in the subject of

Engineering

The University of Electro-Communications

March 2015

# Committee

| | |
|---|---|
| Professor | Hideya Iwasaki |
| | |
| Professor | Rikio Onai |
| Professor | Tetsu Narumi |
| Professor | Masayuki Narumi |
| Associate Professor | Yasuichi Nakayama |

For refereeing this doctoral dissertation, entitled, *Structured Parallel Programming with Trees*.

# Copyright

# Abstract

High-level abstractions for parallel programming are still immature. Computations on complicated data structures such as pointer structures are considered as irregular algorithms. General graph structures, which irregular algorithms generally deal with, are difficult to divide and conquer. Because the divide-and-conquer paradigm is essential for load balancing in parallel algorithms and a key to parallel programming, general graphs are reasonably difficult. However, trees lead to divide-and-conquer computations by definition and are sufficiently general and powerful as a tool of programming. We therefore deal with abstractions of tree-based computations.

Our study has started from Matsuzaki's work on tree skeletons. We have improved the usability of tree skeletons by enriching their implementation aspect. Specifically, we have dealt with two issues. We first have implemented the loose coupling between skeletons and data structures and developed a flexible tree skeleton library. We secondly have implemented a parallelizer that transforms sequential recursive functions in C into parallel programs that use tree skeletons implicitly. This parallelizer hides the complicated API of tree skeletons and makes programmers to use tree skeletons with no burden. Unfortunately, the practicality of tree skeletons, however, has not been improved. On the basis of the observations from the practice of tree skeletons, we deal with two application domains: program analysis and neighborhood computation.

In the domain of program analysis, compilers treat input programs as control-flow graphs (CFGs) and perform analysis on CFGs. Program analysis is therefore difficult to divide and conquer. To resolve this problem, we have developed divide-and-conquer methods for program analysis in a syntax-directed manner on the basis of Rosen's high-level approach. Specifically, we have dealt with data-flow analysis based on Tarjan's formalization and value-graph construction based on a functional formalization.

In the domain of neighborhood computations, a primary issue is locality. A naive parallel neighborhood computation without locality enhancement causes a lot of cache misses. The divide-and-conquer paradigm is known to be useful also for locality enhancement. We therefore have applied algebraic formalizations and a tree-segmenting technique derived from tree skeletons to the locality enhancement of neighborhood computations.

**Quick Overview In Japanese**　並列プログラミングでは高水準の抽象化が十分に確立しておらず，複雑な構造を用いる計算は，不規則アルゴリズムと一括りに扱われている．しかし，木構造を扱う計算は，自然な分割統治が定義できるために，系統的な抽象化を期待できる．そこで本研究は，木を用いる並列計算の抽象化を扱う．まず，松崎による木スケルトンの成果を出発点とし，実装面を充実させることで，その使いやすさを向上させた．しかし，実用性は十分ではなかったので，そこで得られた知見や培った技法を元に，2 つの問題領域に取り組んだ．1 つはプログラム解析領域において，構文主導型分割統治手法を開発した．もう 1 つは，空間上の近傍計算において，キャッシュ効率化手法を提案した．

# Contents

# Acknowledgments

Many people helped me complete my doctoral research. I would like to express my gratitude to them here.

First of all, I would like to thank my supervisor, Professor Hideya Iwasaki, for his patience in providing me with an ideal milieu for research and in guiding me with his educational consideration even in matters of my life as well as research. His great tolerance for me was indispensable for my research activities.

I would also like to express my deepest gratitude to the co-authors of the publications in my doctoral research, Lecturer Akimasa Morihata of The University of Tokyo and Associate Professor Kiminori Matsuzaki of Kochi University of Technology. They were the ones beyond co-authors for me. They always kindly taught me both technical and non-technical perspectives of research and correct attitudes to research. They even gave me their time to discuss my personal concerns with me. They have been exemplary researchers that had had an invaluable influence on me.

I am very grateful to Professor Zhenjiang Hu of National Institute of Informatics, Assistant Professor Kento Emoto of Kyushu Institute of Technology, and Professor Munehiro Takimoto of Tokyo University of Science. Professor Hu always encouraged me in my research with expecting my success, and gave me many opportunities to attend seminars, workshops, conferences even with financial aids. Assistant Professor Emoto's discussions, comments, and criticisms on my studies and presentations were always helpful to me. Professor Takimoto always encouraged me in my studies on data-flow analysis and gave me his time to discuss them with me.

I am very thankful to Associate Professor Tomoharu Ugawa, Associate Professor Keisuke Nakano, and Associate Professor Yoshihiro Oyama. Associate Professor Ugawa's comments on implementation issues on software systems, especially, memory management were insightful and beneficial to me. Associate Professor Nakano's comments on my presentations were correct, reasonable, and helpful to me. Associate Professor Oyama had interest in my observations on parallel computing and gave me his time to discuss them with me.

I am grateful to Associate Professor Miyako Satoh of The University of Electro-Communications for discussions with me on English and for her kind instruction on my linguistic usage. I am thankful to Assistant Professor Kazutaka Matsuda of The University of Tokyo and Yasunao Takano of my colleagues for their talks and discussions with me that were a beneficial, refreshing, or enjoyable time for me.

Last but not least, those whom I cannot name, my heartfelt thanks go to members of the Iwasaki laboratory, ones of the IPL seminar, and many others for their mental stimulation to me.

# Chapter 1

# Introduction

## 1.1 Background and Motivation

In the history of computer science, parallel computing and parallel machines are classic topics. In the research on programming languages, parallel programs (i.e., programs exploiting parallel machines) and parallel programming (i.e., to develop parallel programs) are never new topics. However, the current status of parallel programming is very immature. The most popular language used in practice for describing parallel programs is FORTRAN, which is one of the oldest programming languages, together with MPI, which is a low-level communication construct. Abstractions for parallel programming are very limited and low-level compared with sequential (i.e., non-parallel) programming. Therefore, parallel programming is considered as a research problem for the next 50 years [HPP09].

Automatic parallelization and optimization are closely relevant to parallel programming because these promote high-level programming that is oblivious of low-level issues that compilers can resolve. Automatic parallelization and optimization for index access to arrays in nested for loops were well studied [AK01, BHRS08, GGL12, YFRS13]. For good or ill, they help FORTRAN survive so far. Meanwhile, although accumulative computation is generally difficult to parallelize, methods for automatic parallelization of accumulative linear recursions were developed [MMM$^+$07, MM10, SI11a, FG94, RF00]. In this sense, the foundations of the abstractions for parallel programming on linear structures are now available. However, more complicated structures are difficult to deal with in parallel programming. Foundations for dealing systematically with them have not been solidly established.

Actually, problems/algorithms that involve more complicated structures, e.g., linked structures and sparse representations than random-access arrays are called *irregular* ones in the context of parallel programming[1]. A systematic approach to implementing irregular algorithms is a big issue in parallel programming and a hot topic in recent studies [KBI$^+$09].

The computational structure in irregular problems forms a graph. Since (general) graphs do not have recursive structures, irregular algorithms are not recursively defined in general. This means that irregular algorithms are generally not contained in the divide-and-conquer paradigm. Divide and conquer is, however, essential to parallel computing and has been used as the key notion extensively in parallel programming [BFGS12, FLR98, BCH$^+$94, Ski93]. Irregular problems/algorithms are therefore inherently difficult to deal with in parallel programming.

Instead of general irregular algorithms, we focus on tree-based algorithms. Although tree-based algorithms are usually considered as irregular ones, their computational structures are based on trees but not (general) graphs. Trees have recursive structures and tree-based algorithms utilize this trait. As a result, the large part of a tree-based algorithm is recursively defined and adopts the divide-and-conquer paradigm. Tree-based algorithms are therefore relatively tractable in parallel programming.

Computations with trees are of high importance. Trees are fundamental data structures in functional programming because they by definition match recursive functions. It is no exaggeration to say that computations with trees cover the core part of functional programming. In parallel programming, trees

[1] `http://www.cs.cmu.edu/~scandal/alg/whatis.html`

work as abstractions for load balancing [MM11b, MMHT09, Mor13] and scheduling [LKK13] because they can represent divide-and-conquer structures directly. These facts promise that high-level abstractions and systematic methods for computations with trees are useful for a wide range of applications. In particular, we can even expect that these lead to a versatile method for divide-and-conquer load balancing.

## 1.2 Overview of This Dissertation

In this dissertation, we deal with parallel programming with trees in a divide-and-conquer manner, which leads to good load balancing. Our study has started from Matsuzaki's work [Mat07b] on tree skeletons [Ski96, GCS94], which are divide-and-conquer parallel patterns on trees. Although Matsuzaki's work investigated the theoretical aspect of tree skeletons, it left much room for improvement on the practical aspect, especially usability. We therefore have improved the usability of tree skeletons on two aspects. One is the modularity and flexibility of library implementation [Mat07a]. We have achieved loose coupling between tree data structures and tree skeletons in our library implementation. Our library is consequently more flexible than existing ones. The other is the complicated APIs of tree skeletons. We have developed a parallelizer that transforms sequential recursive functions in C into tree skeleton calls on the basis of the formalization by Matsuzaki at al. [MHT06]. Our parallelizer hides the complicated API of tree skeletons from programmers and enables programmers to use tree skeletons implicitly.

Although we have improved the usability of tree skeletons on the two aspects successfully, programming with tree skeletons is still not useful in practice. We have noticed that a purely structural approach that tree skeletons adopt per se is counter-intuitive and it is important to examine the interpretations of trees and the underlying data of trees. This observation suggests the fundamental importance of application domains. We therefore have determined to distance ourselves from Matsuzaki's work on tree skeletons and focused on two domains: program analysis and spatial computation.

We have dealt with program analysis based on abstract syntax trees (ASTs) because its computational structure is similar to tree skeletons. Specifically, we have dealt with data-flow analysis and value-graph construction, which are the foundations of compiler optimizations. In AST-based approaches, goto/label statements are troublesome because a syntax-directed handling of them is difficult. From the perspective of irregular algorithms, goto/label statements cause computations on general graphs and become an obstacle to divide-and-conquer computations. In fact, compilers use control-flow graphs, which are general directed graphs, as input programs to most kinds of program analysis. As a result, they are difficult to parallelize in a divide-and-conquer manner. On the basis of the notion of Rosen's high-level data-flow analysis [Ros77, Ros80], which does not deal with goto/label statements, we have developed a mostly divide-and-conquer method for data-flow analysis and and value-graph construction to ASTs containing few goto/label statements.

We have dealt with spatial computation based on neighborhood (or simply neighborhood computation) because of its practical importance in various applications. Although this computation is usually trivial to perform in a divide-and-conquer manner, its naive divide-and-conquer implementation is insufficient. Because cache efficiency is of high importance for this computation, a cache-efficient divide-and-conquer implementation like cache-oblivious algorithms [FLPR99, BGS10] is desired. We therefore have investigated cache-efficient divide-and-conquer approaches to two kinds of neighborhood computations. We first have dealt with stencil computation, which is a regular array-based computation but very popular in scientific computing, and have developed a linear algebraic optimization for enhancing its cache efficiency. We secondly have dealt with iterative traversal of space-partitioning trees, which is a typical algorithmic pattern found in $N$-body problems and machine learning, and have developed a skeleton-based technique for enhancing its cache efficiency.

Through the practice of parallel programming in the two domains, we have confirmed our observation on parallel programming with trees.

## 1.3 Contributions and Organization of This Dissertation

This dissertation consists of three parts:

- Part I deals with programming with tree skeletons. This part is overall based on Matsuzaki's work [Mat07b] on tree skeletons (Chapter 2). We deal with problems on the usability of tree skeletons. We present a tree interface to achieve loose coupling between the implementation of trees and that of tree skeletons (Chapter 3). We present a parallelizer for recursive functions in C to enable implicit use of tree skeletons (Chapter 4). We describe observations on programming with trees from limitations of tree skeletons (Chapter 5).
- Part II deals with syntax-directed programming in the domain of high-level program analysis (Chapter 6). On the basis of the notion of Rosen's high-level data-flow analysis [Ros77, Ros80], we present syntax-directed methods for data-flow analysis (Chapter 7) and for value-graph construction (Chapter 8). These methods perform mostly in a divide-and-conquer manner by taming goto/label statements.
- Part III deals with cache-efficient divide-and-conquer programming in the domain of neighborhood computations (Chapter 10). We present a linear algebraic optimization for enhancing the cache complexity of stencil computation (Chapter 11) and present a skeleton-based technique for enhancing the cache complexity of iterative traversal of space-partitioning trees (Chapter 12).

The individual results above have been published or presented. Chapter 3 was published in [SM14a], Chapter 4 was published in [SM13], Chapter 7 was published in [SM14b], Chapter 8 was presented in [Sat14b], Chapter 11 was presented in [SI11b], and Chapter 12 was presented in [Sat14a].

# Part I

# Programming With Tree Skeletons

# Chapter 2

# Tree Skeletons

In this chapter, we introduce tree skeletons [Ski96, GCS94], which are background knowledge for Part I. Refer to Matsuzaki's Ph.D. thesis [Mat07b] for more details.

## 2.1 What Is the Skeleton?

Algorithmic skeletons [Col89] (or simply skeletons) are patterns of parallel computing. Skeletons are high-level abstractions for parallel programming. By composing skeletons, we can develop parallel programs in a high-level and structured manner [DGTY95].

There are roughly two kinds of skeletons [RG02, GVL10]: task-parallel ones and data-parallel ones. Task-parallel skeletons, or task skeletons for short, represent and manipulate tasks. Typical task skeletons are pipe and farm, which respectively correspond to serial composition of tasks and parallel composition. A structure that task skeletons construct is an analogy to a circuit. Task skeletons exploit parallelism of computations represented by this circuit; e.g., pipe exploits parallelism in pipelining. We use the term *parallelism* as a synonym for independence of (sub)computations. Although serial compositions by pipe do not have so-called parallelism in their circuits, pipelined computations on their circuits have parallelism.

Data-parallel skeletons manipulate collections in parallel. More precisely, they are collective parallel operations on a certain data structure. The most popular data-parallel skeleton is map: it takes a unary function and a collection, and yields a new collection consisting of the results that we obtain by applying the given function to each element of the given collection. map can be defined over various data structures. For example, if map takes a list and yields a new list, this map is called a list skeleton (i.e., a collective parallel operation over lists). Data-parallel skeletons are thus classified by their target data structures.

We deal with data-parallel skeletons in this dissertation. Readers may consider task skeletons to be more general than data-parallel skeletons. This is true in the sense that task skeletons are independent of target data structures. Although individual data-parallel skeletons have more specific purposes than task skeletons, data-parallel skeletons per se are no less versatile. The difference between both is essentially in formalization of computation. When we use data-parallel skeletons, we first formalize target computation as a data structure. For example, if the specification of target computation is a linear recursion (or, a sequence of functions), we may use a list(s) to formalize it. We then use list skeletons to describe the specification as a parallel program. That is, data-parallel skeletons do not always handle data structures that are input/output in target computations. Formalization with data structures is essential for programming with data-parallel skeletons. This introduces rich computational structures into specifications. In this sense, programming with data-parallel skeletons is more structured than doing with task skeletons, and therefore we deal with it.

## 2.2 Formalization with Data Structures

To explain the importance of computational structures formalized by data structures, we briefly introduce lists and list skeletons [Ski93] based on functional programming. We write a list by enumerating elements with commas and enclosing them with brackets; e.g., $[0, 1, 2]$. We call a list in this form a *list literal*.

We first describe a notation for data types of trees in this dissertation. We use a restricted form of context free grammars for defining a data type as a nonterminal symbol. A lowercase term (e.g., $\alpha$ and $x$) denotes a metavariable. A capitalized term that is not defined in grammars, i.e., does not appear in the left-hand side of any production rule, denotes a terminal symbol, and a capitalized term that is defined in grammars denotes a nonterminal symbol that denotes a data type. Terminal symbols may have argument lists like functions to specify associated metavariables and symbols. The production rules of any nonterminal symbol can be distinguished from its leftmost terminal symbol as in LL grammars. Pattern matching on the data type is therefore deterministic.

The data type of lists is usually defined as the following grammar:

$$\begin{aligned} \mathit{List}_\alpha &= \mathit{Cons}(x, \mathit{List}_\alpha), \quad \text{where } x \text{ is of type } \alpha, \\ \mathit{List}_\alpha &= \mathit{Nil}. \end{aligned}$$

Here, $x$ denotes a metavariable over list elements. *Nil* denotes the empty list (i.e., $[]$). $\mathit{Cons}(x, y)$ denotes a list whose head is $x$ and whose tail is $y$; e.g., $\mathit{Cons}(a_1, [a_2, a_3]) = [a_1, a_2, a_3]$. This is called the *cons list*. For example, a list (literal) $[a_1, a_2, a_3]$ is a synonym of the unique cons list:

$$\mathit{Cons}(a_1, \mathit{Cons}(a_2, \mathit{Cons}(a_3, \mathit{Nil}))).$$

The data type of lists can be also defined as the following grammar:

$$\begin{aligned} \mathit{JList}_\alpha &= \mathit{Join}(\mathit{JList}_\alpha, \mathit{JList}_\alpha), \\ \mathit{JList}_\alpha &= \mathit{Singleton}(x), \quad \text{where } x \text{ is of type } \alpha, \\ \mathit{JList}_\alpha &= \mathit{Nil}. \end{aligned}$$

$\mathit{Singleton}(x)$ denotes a singleton list (i.e., $[x]$). $\mathit{Join}(x, y)$ denotes the concatenation of two lists $x$ and $y$; $\mathit{Join}([a_1], [a_2, a_3]) = [a_1, a_2, a_3]$. This is called the *join list*. In contrast to the cons list, $[a_1, a_2, a_3]$ can be represented by, for example, the following four join lists:

$$\begin{aligned} &\mathit{Join}(\mathit{Singleton}(a_1), \mathit{Join}(\mathit{Singleton}(a_2), \mathit{Singleton}(a_3))), \\ &\mathit{Join}(\mathit{Join}(\mathit{Singleton}(a_1), \mathit{Singleton}(a_2)), \mathit{Singleton}(a_3)), \\ &\mathit{Join}(\mathit{Join}(\mathit{Singleton}(a_1), \mathit{Singleton}(a_2)), \mathit{Join}(\mathit{Nil}, \mathit{Singleton}(a_3))), \text{ and} \\ &\mathit{Join}(\mathit{Singleton}(a_1), (\mathit{Join}(\mathit{Singleton}(a_2), \mathit{Join}(\mathit{Singleton}(a_3), \mathit{Nil})))). \end{aligned}$$

We can understand this equivalence from the algebraic property of the interpretation of *Join*, i.e., the concatenation of two lists. For example, we can define a concatenation operator $+\!\!+$ over lists by using list literals as follows:

$$\begin{aligned} [a_1, \ldots, a_n] +\!\!+ [b_1, \ldots, b_m] &= [a_1, \ldots, a_n, b_1, \ldots, b_m], \\ x +\!\!+ [] &= x, \\ [] +\!\!+ x &= x, \end{aligned}$$

where $m, n \geqslant 1$ and $x$ denotes a list. It is important that $+\!\!+$ is associative; for any $x$, $y$, and $z$, $x +\!\!+ (y +\!\!+ z) = (x +\!\!+ y) +\!\!+ z$, and that $[]$ is the identity of $+\!\!+$. By using $+\!\!+$, we can confirm the equivalence among the four join lists as follows:

$$\begin{aligned} [a_1] +\!\!+ ([a_2] +\!\!+ [a_3]) &= [a_1, a_2, a_3], \\ ([a_1] +\!\!+ [a_2]) +\!\!+ [a_3] &= [a_1, a_2, a_3], \\ ([a_1] +\!\!+ [a_2]) +\!\!+ ([] +\!\!+ [a_3]) &= [a_1, a_2, a_3], \\ [a_1] +\!\!+ ([a_2] +\!\!+ ([a_3] +\!\!+ [])) &= [a_1, a_2, a_3]. \end{aligned}$$

This means that the four join lists are contained in the same equivalence class modulo interpreting *Join* as ++. From the perspective of data structures, the associativity of ++ and its identity [] enable us to transform join lists in the same equivalence class.

A high degree of freedom in structure among lists in an equivalence class is of primary importance for parallel programming. We can improve structural parallelism of join lists by balancing them. For example, we consider map, which can be defined with list literals as follows:

$$\begin{aligned} \mathsf{map}(f, [a_1, \ldots, a_n]) &= [f(a_1), \ldots, f(a_n)], \quad \text{where } n \geqslant 1, \\ \mathsf{map}(f, []) &= []. \end{aligned}$$

We can define map over $List_\alpha$ as

$$\begin{aligned} \mathsf{map}(f, \mathit{Cons}(x, y)) &= \mathit{Cons}(f(x), \mathsf{map}(f, y)), \\ \mathsf{map}(f, \mathit{Nil}) &= \mathit{Nil}. \end{aligned}$$

We can also define map over $JList_\alpha$ as

$$\begin{aligned} \mathsf{map}(f, \mathit{Join}(x, y)) &= \mathit{Join}(\mathsf{map}(f, x), \mathsf{map}(f, y)), \\ \mathsf{map}(f, \mathit{Singleton}(x)) &= \mathit{Singleton}(f(x)), \\ \mathsf{map}(f, \mathit{Nil}) &= \mathit{Nil}. \end{aligned}$$

Then, let us consider $\mathsf{map}(f, [a_1, a_2, a_3])$. The recursions for the cons list and the fourth join list are almost the same because the shapes of both list representations are equivalent. Their recursion depth is 4. The recursion for the third join list is, however, different from them. It halves a given list recursively and therefore its recursion depth is 3. In general, given a list of length $n$, map over $List_\alpha$ costs $\mathrm{O}(n)$ parallel steps but map over $JList_\alpha$ costs $\mathrm{O}(\lg n)$ parallel steps. Balancing join lists leads to load balancing of list skeletons. This is why the join list is seen as a parallel implementation of the list.

The definition of map over $JList_\alpha$ does not specify load balancing itself but simply exploits structural parallelism of a given join list. The interpretation of $JList_\alpha$ expresses room for load balancing on the basis of the associativity of the interpretation of *Join* (i.e., ++).

The algebraic properties of the interpretation of $JList_\alpha$ affect the specifications of list skeletons. For example, we consider reduce defined as follows:

$$\begin{aligned} \mathsf{reduce}(\oplus, \iota_\oplus, [a_1, \ldots, a_n]) &= a_1 \oplus \cdots \oplus a_n, \quad \text{where } n \geqslant 1, \\ \mathsf{reduce}(\oplus, \iota_\oplus, []) &= \iota_\oplus. \end{aligned}$$

reduce has the algebraic conditions that $\oplus$ is associative and $\iota_\oplus$ is the identity of $\oplus$. We can understand it from the definition of reduce over $JList_\alpha$:

$$\begin{aligned} \mathsf{reduce}(\oplus, \iota_\oplus, \mathit{Join}(x, y)) &= \mathsf{reduce}(\oplus, \iota_\oplus, x) \oplus \mathsf{reduce}(\oplus, \iota_\oplus, y), \\ \mathsf{reduce}(\oplus, \iota_\oplus, \mathit{Singleton}(x)) &= x, \\ \mathsf{reduce}(\oplus, \iota_\oplus, \mathit{Nil}) &= \iota_\oplus. \end{aligned}$$

reduce thus interprets *Join* as $\oplus$ and *Nil* as $\iota_\oplus$. The join lists in an equivalence class must be equivalent for every list skeleton. For example, $\mathsf{reduce}(\oplus, \iota_\oplus, [a_1, a_2, a_3])$ have to yield the same result among the four join lists. To ensure the equality of their results, the algebraic conditions on $\oplus$ and $\iota_\oplus$ are necessary. They are derived from the interpretation of $JList_\alpha$. From the perspective of list skeletons, such algebraic conditions are the representation of their parallelism.

The definitions of data-parallel skeletons are oblivious of parallel computing and therefore data-parallel skeletons enable high-level parallel programming. Parallelism of data-parallel skeletons is described in target data structures (and more specifically, the interpretations of data types). Formalization with data structures is therefore crucial for high-level parallel programming.

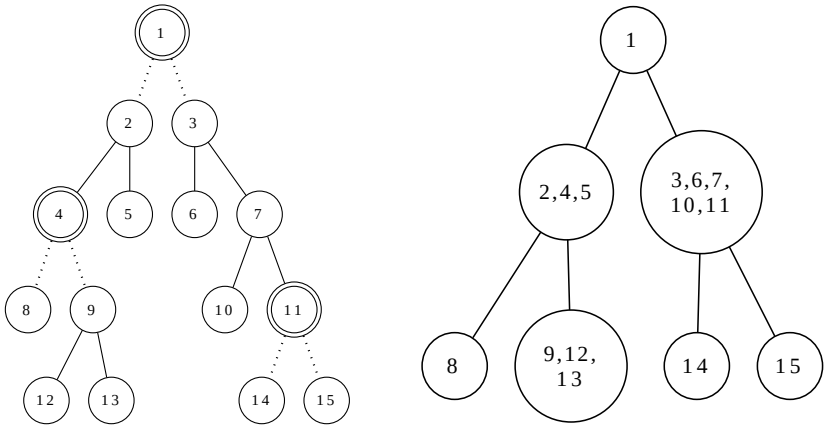



Figure 2.1: Illustration of a segmented tree. On left is view decomposed into its local trees, where double-circle nodes denote hole nodes and dotted edge denote missing edges; on right is view of its global tree.

## 2.3 Divide and Conquer on Segmented Trees

### 2.3.1 Segmented Trees

Tree skeletons [Ski96, GCS94, Mat07b] deal with full binary trees. We define full binary trees to formalize tree skeletons as the following grammar:

$$\begin{aligned} \mathit{BinTree}_\alpha &= \mathit{Branch}(x, \mathit{BinTree}_\alpha, \mathit{BinTree}_\alpha), \\ \mathit{BinTree}_\alpha &= \mathit{Leaf}(x), \end{aligned}$$

where $x$ denotes a metavariable over values of type $\alpha$. We call the value of $x$ a *payload value*. *Branch* and *Leaf* denote the kinds of nodes; we call them *node symbols*.

Although $\mathit{BinTree}_\alpha$ is seemingly similar to $\mathit{JList}_\alpha$, both interpretations are different. Unlike *Join*, the interpretation of *Branch* is generally not associative because rotations are not applicable to every tree. We cannot balance trees of type $\mathit{BinTree}_\alpha$, preserving their meaning. $\mathit{BinTree}_\alpha$ therefore does not guarantee load balancing of tree skeletons. For example, a tree of type $\mathit{BinTree}_\alpha$ may form a linear structure like the cons list. The recursion depth for such a tree will be $\mathrm{O}(n)$, where $n$ is the number of nodes. To achieve *robust* load balancing such that it guarantees asymptotic speedup regardless of input trees, we have to consider restructuring of arbitrarily shaped trees.

Like join lists for list skeletons, we use *segmented trees*[1] for tree skeletons. A segmented tree is a tree segmented into its subtrees and its one-hole contexts. Figure 2.1 illustrates a segmented tree. A one-hole context is a subtree that loses the descendant subtrees of one node. This node, i.e., a parent of missing subtrees, is called a *hole node*. For example, $\{1\}$ and $\{2, 4, 5\}$ on the left of Figure 2.1 are one-hole contexts and 1 and 4 are hole nodes. A hole node is an internal node in the whole tree but pretends to be a leaf node in the one-hole context. Every segment (i.e., subtree or one-hole context) of a segmented tree is a tree and we call it a *local tree*. We also call the tree structure of all segments in a segmented tree *a global tree*, where the node of a global tree is a segment. Therefore, the nodes of each segment in Figure 2.1 reduces to a single node in the view of its global tree.

[1] The same term was not used in [Mat07b] but an equivalent notion was used.

We can define segmented trees as the following $SegTree_\alpha$:

$$
\begin{aligned}
SegTree_\alpha &= Branch(Context_\alpha, SegTree_\alpha, SegTree_\alpha),\\
SegTree_\alpha &= Leaf(BinTree_\alpha),\\
Context_\alpha &= Branch_1(x, Context_\alpha, BinTree_\alpha),\\
Context_\alpha &= Branch_2(x, BinTree_\alpha, Context_\alpha),\\
Context_\alpha &= Hole(x).
\end{aligned}
$$

Here, $BinTree_\alpha$ denotes the subtrees of an original tree, $Context_\alpha$ denotes the one-hole contexts, and $Hole(x)$ denotes the hole nodes. As seen from the grammar above, the root-to-hole path in a one-hole context is the main difference between subtrees and one-hole contexts. Note that the formalization above of segmented trees is inherently similar to ternary trees [Mat07b, MM11b]. By following the grammar above, we describe the segmented tree shown in Figure 2.1 as

$$
\begin{aligned}
&Branch(c_1, Branch(c_2, Leaf(s_1), Leaf(s_2)), Branch(c_3, Leaf(s_3), Leaf(s_4))),\\
&\text{where } c_1 = Hole(1),\\
&\qquad c_2 = Branch_1(2, Hole(4), Leaf(3)),\\
&\qquad c_3 = Branch_2(3, Leaf(6), Branch_2(7, Leaf(10), Hole(11))),\\
&\qquad s_1 = Leaf(8),\\
&\qquad s_2 = Branch(9, Leaf(12), Leaf(13)),\\
&\qquad s_3 = Leaf(14),\\
&\qquad s_4 = Leaf(15).
\end{aligned}
$$

### 2.3.2 Properties of Segmented Trees

Segmented trees have two important properties. The first is that the operations of parallel tree contraction [Rei93] guarantee parallel reduction of every segment. The second is that the $m$-bridging technique [Rei93] enables us to partition an arbitrarily shaped tree into segments of almost balanced sizes. The details of tree contraction operations and $m$-bridging shall be described later.

The segmented trees derived from a tree are contained in an equivalence class modulo appropriate interpretations of tree contraction operations. Tree skeletons can exploit this degree of freedom in structure of segmented trees. If we use segmented trees whose segment sizes are balanced, tree skeletons achieve an excellent load balancing. Note that any tree can be seen as a segmented tree that consists of the single segment. The equivalence class of the segmented trees derived from a given tree therefore contains the given tree.

Along the structure of a segmented tree, the divide-and-conquer computation over a segmented tree is two-tiered; it consists of computation over a local tree and computation over a global tree. A segmented tree is therefore well-suited to distributed memory. By laying an entire segment upon the local memory of a processor, we can perform the entire computation over each local tree in parallel. In this case, the computation over a global tree requires a collective communication on distributed-memory machines. This communication is efficient because a global tree is much smaller than its original tree.

## 2.4 Tree Contraction Operations

In this section, we briefly describe tree contraction operations on the basis of Reference [Rei93].

There are two basic operations: Rake and Compress. The Rake operation is to contract a leaf and its parent to an internal node. The Compress operation is to contract an internal siblingless node and its parent to an internal node. Figure 2.2 illustrates Rake and Compress. The original tree contraction algorithm by Miller and Reif [MR85] uses Rake and Compress.

Although Rake and Compress are intuitive, an intermediate result in a series of these applications to a full binary tree becomes a general (i.e., non-full) binary tree. However, if we perform a pair of

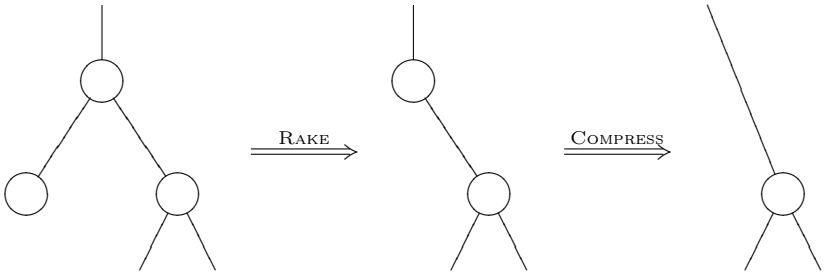



Figure 2.2: Rake operation and Compress operation.

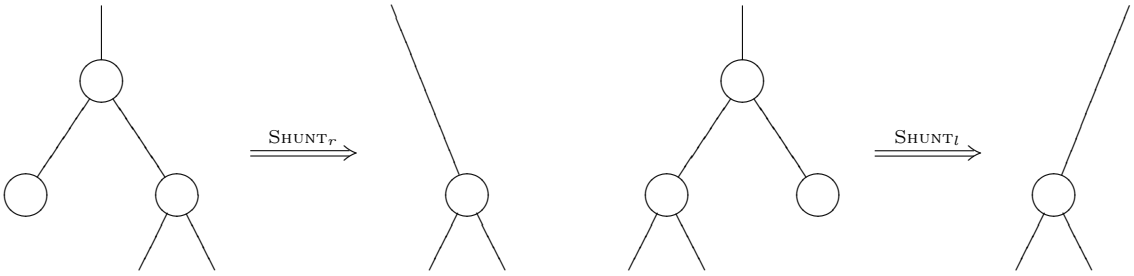



Figure 2.3: Shunt operations.

applications of Rake and Compress at once as illustrated in Figure 2.2, the intermediate results from full binary trees remain full binary trees. This pair of applications of Rake and Compress is called the Shunt operation. Specifically, the Shunt operation is to contract the siblings of a leaf node and an internal node, and their parent to an internal node, as illustrated in Figure 2.3. We can consider two symmetrical instances of the Shunt operation for binary trees. In this dissertation, Shunt$_r$ means the Shunt operation that applies Rake to a left leaf and Shunt$_l$ means the Shunt operation that applies Rake to a right leaf. The cost-optimal tree contraction algorithm on EREW PRAM by Abrahamson et al. [ADKP89] uses the Shunt operation.

## 2.5 Definitions of Tree Skeletons

Parallel tree contraction algorithms guarantee asymptotically linear speedup for arbitrarily shaped trees. Because of this excellent algorithmic property, existing formalizations [Ski96, GCS94, Mat07b, MMHT09] of tree skeletons are based upon parallel tree contraction algorithms. However, in this dissertation, we do not deal with these algorithms themselves because they are not very useful for implementation. If we use optimal algorithms, the parallel time complexity of the divide-and-conquer computation over a global tree improves from $O(p)$ to $O(\lg p)$ but synchronization/communication steps increase in practice. We therefore adopt sequential manners for computations on global trees. Note that bulk synchronous communication steps are $O(1)$ and the amount of communication data is $O(p)$ in a collective communication for the computation over a global tree on distributed-memory machines.

We assume that the sizes of the segments of a given segmented tree are sufficiently balanced. This is not a responsibility of tree skeletons. We have to give tree construction operations appropriate interpretations for using tree skeletons. The appropriate interpretations here mean that the results of a tree skeleton equal for any segmented tree in the equivalence class. This requirement is represented as algebraic conditions on the parameters of tree skeletons, as in the reduce list skeleton.

We introduce three representative tree skeletons reduce, uAcc, and dAcc. The sequential definition of

reduce is the following bottom-up reduction:

$$
\begin{aligned}
&\mathsf{reduce} : (\beta \times \alpha \times \beta \to \beta) \times (\alpha \to \beta) \times \mathit{BinTree}_\alpha \to \beta \\
&\mathsf{reduce}(k_B, k_L, \mathit{Branch}(x, l, r)) = k_B(\mathsf{reduce}(k_B, k_L, l), x, \mathsf{reduce}(k_B, k_L, r)), \\
&\mathsf{reduce}(k_B, k_L, \mathit{Leaf}(x)) = k_L(x).
\end{aligned}
$$

The parallel definition of reduce takes, instead of a simple tree of type $\mathit{BinTree}_\alpha$, a segmented tree of type $\mathit{SegTree}_\alpha$. Recall that for the divide and conquer over a segment tree, the appropriate interpretations of tree contraction operations must be given. The parallel definition of reduce therefore requires additional parameter operators: $\tau$, $\phi$, $\psi_l$, and $\psi_r$. $\psi_l$ and $\psi_r$ respectively correspond to $\textsc{Shunt}_l$ and $\textsc{Shunt}_r$. To conform the reduce of a segmented tree to that of a simple tree, the parallel definition of reduce necessitates some algebraic conditions among parameter operators. The parallel definition of reduce is:

$$
\begin{aligned}
&\mathsf{reduce} : (\beta \times \alpha \times \beta \to \beta) \times (\alpha \to \beta) \times (\alpha \to \gamma) \times (\beta \times \gamma \times \beta \to \beta) \\
&\qquad \times (\gamma \times \gamma \times \beta \to \gamma) \times (\beta \times \gamma \times \gamma \to \gamma) \times \mathit{SegTree}_\alpha \to \beta \\
&\mathsf{reduce}(k_B, k_L, \tau, \phi, \psi_l, \psi_r, t) = \mathit{red}(t), \\
&\text{where } \mathit{red}(\mathit{Branch}(c, l, r)) = \phi(\mathit{red}(l), \mathit{redCxt}(c), \mathit{red}(r)), \\
&\qquad \mathit{red}(\mathit{Leaf}(t)) = \mathsf{reduce}(k_B, k_L, t), \\
&\qquad \mathit{redCxt}(\mathit{Branch}_1(x, l, r)) = \psi_l(\mathit{redCxt}(l), \tau(x), \mathsf{reduce}(k_B, k_L, r)), \\
&\qquad \mathit{redCxt}(\mathit{Branch}_2(x, l, r)) = \psi_r(\mathsf{reduce}(k_B, k_L, l), \tau(x), \mathit{redCxt}(r)), \\
&\qquad \mathit{redCxt}(\mathit{Hole}(x)) = \tau(x), \\
&\qquad \{\text{Algebraic conditions}\} \\
&\qquad k_B(l, x, r) = \phi(l, \tau(x), r), \\
&\qquad \phi(\phi(l', y', r'), y, r) = \phi(l', \psi_l(y', y, r), r'), \\
&\qquad \phi(l, y, \phi(l', y', r')) = \phi(l', \psi_r(l, y, y'), r').
\end{aligned}
$$

Although the parallel definition above is seemingly involved, it simply interprets node symbols by using given operators, as in the sequential definition. The definition above formalizes reduce over $\mathit{SegTree}_\alpha$ but does not describe its actual computation. The actual computation of reduce consists of two phases. The first is the bottom-up reduction of every local tree; the second is the bottom-up reduction of a global tree. In the first phase, we use $k_B$ and $k_L$ for reducing complete subtrees. As a result, for a one-hole context, the root-to-hole spine remains. We use $\tau$, $\psi_l$, and $\psi_r$ for reducing such spines. We use $\phi$ for reducing a global tree.

uAcc (upward accumulation) is an accumulative version of reduce, i.e., the following accumulation:

$$
\begin{aligned}
&\mathsf{uAcc} : (\beta \times \alpha \times \beta \to \beta) \times (\alpha \to \beta) \times \mathit{BinTree}_\alpha \to \mathit{BinTree}_\beta \\
&\mathsf{uAcc}(k_B, k_L, \mathit{Branch}(x, l, r)) = \mathit{Branch}(k_B(\mathit{root}(l'), x, \mathit{root}(l')), l', r'), \\
&\quad \text{where } l' = \mathsf{uAcc}(k_B, k_L, l), \\
&\qquad\quad r' = \mathsf{uAcc}(k_B, k_L, r), \\
&\qquad\quad \mathit{root}(\mathit{Branch}(x, l, r)) = x, \\
&\qquad\quad \mathit{root}(\mathit{Leaf}(x)) = x, \\
&\mathsf{uAcc}(k_B, k_L, \mathit{Leaf}(x)) = \mathit{Leaf}(k_L(x)).
\end{aligned}
$$

uAcc constructs a tree that has the same shaped as a given tree and whose payload values are intermediate values in reduce. The payload value of the root of a resultant tree equals to the result of reduce. The operators given to uAcc and the algebraic conditions on them are the same as those of reduce. The

parallel definition of uAcc is:

$$
\begin{aligned}
&\mathsf{uAcc} : (\beta \times \alpha \times \beta \to \beta) \times (\alpha \to \beta) \times (\alpha \to \gamma) \times (\beta \times \gamma \times \beta \to \beta) \\
&\qquad \times (\beta \times \gamma \times \gamma \to \gamma) \times (\gamma \times \gamma \times \beta \to \gamma) \times \mathit{SegTree}_\alpha \to \mathit{SegTree}_\beta \\
&\mathsf{uAcc}(k_B, k_L, \tau, \phi, \psi_l, \psi_r, \mathit{Branch}(c, l, r)) = \mathit{Branch}(c', l', r'), \\
&\quad \text{where } c' = \mathit{updPath}((\mathit{segRoot}(l'), \mathit{segRoot}(l')), \mathit{uAccCtx}(c)), \\
&\qquad l' = \mathsf{uAcc}(k_B, k_L, \tau, \phi, \psi_l, \psi_r, l), \\
&\qquad r' = \mathsf{uAcc}(k_B, k_L, \tau, \phi, \psi_l, \psi_r, r), \\
&\qquad \mathit{uAccCtx}(\mathit{Branch}_1(x, l, r)) = \mathit{Branch}_1(x'', l'', r''), \\
&\qquad\quad \text{where } x'' = \psi_l(\mathit{ctxRoot}(l''), \tau(x), \mathit{ctxRoot}(r'')), \\
&\qquad\qquad l'' = \mathit{uAccCtx}(l), \\
&\qquad\qquad r'' = \mathsf{uAcc}(k_B, k_L, r), \\
&\qquad \mathit{uAccCtx}(\mathit{Branch}_2(x, l, r)) = \mathit{Branch}_2(x'', l'', r''), \\
&\qquad\quad \text{where } x'' = \psi_r(\mathit{ctxRoot}(l''), \tau(x), \mathit{ctxRoot}(r'')), \\
&\qquad\qquad l'' = \mathsf{uAcc}(k_B, k_L, l), \\
&\qquad\qquad r'' = \mathit{uAccCtx}(r), \\
&\qquad \mathit{uAccCtx}(\mathit{Hole}(x)) = \mathit{Hole}(\tau(x)), \\
&\qquad \mathit{updPath}((x_l, x_r), \mathit{Branch}_1(x, l, r)) = \mathit{Branch}_1(\phi(x_l, x, x_r), l, r), \\
&\qquad \mathit{updPath}((x_l, x_r), \mathit{Branch}_2(x, l, r)) = \mathit{Branch}_2(\phi(x_l, x, x_r), l, r), \\
&\qquad \mathit{updPath}((x_l, x_r), \mathit{Hole}(x)) = \mathit{Hole}(\phi(x_l, x, x_r)), \\
&\qquad \mathit{segRoot}(\mathit{Branch}(c, l, r)) = \mathit{ctxRoot}(c), \\
&\qquad \mathit{segRoot}(\mathit{Leaf}(t)) = \mathit{root}(t), \\
&\qquad \mathit{ctxRoot}(\mathit{Branch}_1(x, l, r)) = x, \\
&\qquad \mathit{ctxRoot}(\mathit{Branch}_2(x, l, r)) = x, \\
&\qquad \mathit{ctxRoot}(\mathit{Hole}(x)) = x, \\
&\mathsf{uAcc}(k_B, k_L, \tau, \phi, \psi_l, \psi_r, \mathit{Leaf}(t)) = \mathit{Leaf}(\mathsf{uAcc}(k_B, k_L, t)), \\
&\text{where } \{\text{Algebraic conditions}\} \\
&\qquad k_B(l, x, r) = \phi(l, \tau(x), r), \\
&\qquad \phi(\phi(l', y', r'), y, r) = \phi(l', (\psi_l(y', y, r), r')), \\
&\qquad \phi(l, y, \phi(l', y', r')) = \phi(l', (\psi_r(l, y, y'), r')).
\end{aligned}
$$

Here, one-hole contexts that *uAccCtx* yields and *updPath* consumes are intermediate results defined as the following grammar:

$$
\begin{aligned}
&\mathit{Context}_{\beta,\gamma} = \mathit{Branch}_1(x, \mathit{Context}_{\beta,\gamma}, \mathit{BinTree}_\beta), \\
&\mathit{Context}_{\beta,\gamma} = \mathit{Branch}_2(x, \mathit{BinTree}_\beta, \mathit{Context}_{\beta,\gamma}), \\
&\mathit{Context}_{\beta,\gamma} = \mathit{Hole}(x),
\end{aligned}
$$

where $x$ denotes a metavariable over values of type $\gamma$. The definition above merely formalizes uAcc over $\mathit{SegTree}_\alpha$. The actual computation of uAcc consists of three phases. The first is the bottom-up accumulation of every local tree; the second is the bottom-up accumulation of a global tree; the third is the bottom-up accumulation of all root-to-hole paths of every one-hole context by using the payload values of the missing children of the hole nodes in a resultant tree.

The sequential definition of dAcc (downward accumulation) is the following top-down accumulation:

$$
\begin{aligned}
&\mathsf{dAcc} : (\beta \times \alpha \to \beta) \times (\beta \times \alpha \to \beta) \times \beta \times \mathit{BinTree}_\alpha \to \mathit{BinTree}_\beta \\
&\mathsf{dAcc}(g_l, g_r, e, \mathit{Branch}(x, l, r)) = \mathit{Branch}(e, \mathsf{dAcc}(g_l, g_r, g_l(e, x), l), \mathsf{dAcc}(g_l, g_r, g_r(e, x), r)), \\
&\mathsf{dAcc}(g_l, g_r, e, \mathit{Leaf}(x)) = \mathit{Leaf}(e).
\end{aligned}
$$

The parallel definition of $\mathsf{dAcc}$ also requires additional parameter operations: $\tau_l$, $\tau_r$, $\rho$, and $\oplus$. $\oplus$ corresponds to COMPRESS and $\rho$ corresponds to RAKE. It also necessitates algebraic conditions among parameter operators. The parallel definition of $\mathsf{dAcc}$ is:

$$
\begin{aligned}
&\mathsf{dAcc} : (\beta \times \alpha \to \beta) \times (\beta \times \alpha \to \beta) \times \beta \times (\alpha \to \gamma) \times (\alpha \to \gamma) \\
&\qquad \times (\beta \times \gamma \to \beta) \times (\gamma \times \gamma \to \gamma) \times \mathit{SegTree}_\alpha \to \mathit{SegTree}_\beta \\
&\mathsf{dAcc}(g_l, g_r, e, \tau_l, \tau_r, \rho, \oplus, \mathit{Branch}(c, l, r)) = \mathit{Branch}(c', l', r'), \\
&\quad \text{where } c' = \mathit{dAccCtx}(e, c), \\
&\qquad\quad l' = \mathsf{dAcc}(g_l, g_r, \rho(e, \mathit{redPath}(\tau_l, c)), \tau_l, \tau_r, \rho, \oplus, l), \\
&\qquad\quad r' = \mathsf{dAcc}(g_l, g_r, \rho(e, \mathit{redPath}(\tau_r, c)), \tau_l, \tau_r, \rho, \oplus, r), \\
&\qquad\quad \mathit{redPath}(\tau, \mathit{Branch}_1(x, l, r)) = \tau_l(x) \oplus \mathit{redPath}(\tau, l), \\
&\qquad\quad \mathit{redPath}(\tau, \mathit{Branch}_2(x, l, r)) = \tau_r(x) \oplus \mathit{redPath}(\tau, r), \\
&\qquad\quad \mathit{redPath}(\tau, \mathit{Hole}(x)) = \tau(x), \\
&\qquad\quad \mathit{dAccCtx}(e, \mathit{Branch}_1(x, l, r)) = \mathit{Branch}_1(e, \mathit{dAccCtx}(g_l(e, x), l), \mathsf{dAcc}(g_l, g_r, g_r(e, x), r)), \\
&\qquad\quad \mathit{dAccCtx}(e, \mathit{Branch}_2(x, l, r)) = \mathit{Branch}_2(e, \mathsf{dAcc}(g_l, g_r, g_l(e, x), l), \mathit{dAccCtx}(g_r(e, x), r)), \\
&\qquad\quad \mathit{dAccCtx}(e, \mathit{Hole}(x)) = \mathit{Hole}(e), \\
&\mathsf{dAcc}(g_l, g_r, e, \tau_l, \tau_r, \rho, \oplus, \mathit{Leaf}(t)) = \mathit{Leaf}(\mathsf{dAcc}(g_l, g_r, e, t)), \\
&\text{where } \{\text{Algebraic conditions}\} \\
&\qquad g_l(e, x) = \rho(e, \tau_l(x)), \\
&\qquad g_r(e, x) = \rho(e, \tau_r(x)), \\
&\qquad \rho(\rho(e, y), y') = \rho(e, y \oplus y').
\end{aligned}
$$

The definition above merely formalizes $\mathsf{dAcc}$ over $\mathit{SegTree}_\alpha$. The actual computation of $\mathsf{dAcc}$ consists of three phases. The first is the reduction of the root-to-hole paths of every one-hole context by using $\tau_l$, $\tau_r$, and $\oplus$; the second is the top-down accumulation of a global tree by using $\rho$; the third is the top-down accumulation of every segment by using a given $e$ or the payload values of the hole nodes in a resultant tree as initial values.

# Chapter 3

# Interface Between Data Structures and Skeletons

This chapter has a content almost identical to our publication [SM14a].

## 3.1 Introduction

Since parallel machines are widespread, parallel computing can be ubiquitous. Parallel programming is, however, still an expensive task even for expert programmers. To make parallel programming be cheap and next-door, we require high-level abstractions for parallel computing.

Skeletons [**?**, RG02] are patterns of parallel computing. Data-parallel skeletons are ones classified by their target data structures, e.g., lists [Ski93], matrices [EHKT07], and trees [Ski96, GCS94]. These computational patterns denote high-level specifications over target data structures. The data-parallel skeletons therefore provide a high-level abstraction for parallel computing over target data structures.

Data-parallel skeleton libraries provide a pair of the implementation of a data structure and that of a set of skeletons over it. This pair is tightly coupled because parallel access to such a data structure relies on its implementation. However, loose coupling between the implementations of data structures and those of the operations over them is important for modularity and flexibility. For example, C++ STL achieves such loose coupling. Most sequence operations defined in `<algorithm>` operate sequence containers uniformly, regardless of their implementations, e.g., a variable-length array `<vector>` and a doubly-linked list `<list>`, where these have different cost models on access. C++ STL enables us to select various implementations of containers that fit use cases. Loose coupling between skeletons and data structures in skeleton libraries will enable us to select implementations of a data structure.

Selection of implementation is actually valuable for tree skeletons. Because tree data are inherently non-uniform, the implementations of tree structures often require taming their non-uniformity for tree skeletons. For example, a full binary tree, whose every internal node has exactly two children, is required for the brevity of the APIs of tree skeletons. To use tree skeletons, we have to transform non-full binary trees into full binary trees by filling missing leaf nodes with dummy nodes. This transformation is obviously undesirable overhead. We can elude this overhead if we can select an implementation of binary trees that pretends to be a full binary tree by returning a common dummy node when missing leaf nodes are demanded.

A similar situation can be found in XML processing. DOM trees derived from XML documents are ranked unbounded-degree trees. In XML processing by using tree skeletons [Ski97, NEM+07], input DOM trees are preprocessed into full binary trees by inserting internal nodes and dummy leaf nodes. It is valuable to elude this preprocessing by using an implementation of DOM trees that pretends to be full binary trees. In addition, it is known that the bracket structures of XML documents are worth preserving for efficient parallel computing [KME07]. An appropriate implementation of tree structures greatly relies on their input/output formats. A general-purpose yet efficient implementation of trees is extremely difficult.

Since we cannot uniquely define an efficient implementation of tree structures for tree skeletons, flexible switching of instances of tree-structure implementation is desirable. However, as in usual implementations of data-parallel skeletons, the existing ones of tree skeletons are tightly coupled with tree-structure implementation and do not support such flexible switching.

To resolve this problem, we have designed an interface between tree skeletons and tree-structure implementation on the basis of iterators. Our interface clarifies the requirements of tree-structure implementation and supports loose coupling. We have implemented our interface on the basis of C++ templates and implemented tree skeletons [Mat07a] in SkeTo[1] by using our interface.

We have used C++ templates extensively for our generic implementation. Enormous unreadable error messages at compile time are a known problem with such template-based implementations [ME10]. Our new implementation of tree skeletons tames error messages by using a type-checking technique.

This chapter presents our design and implementation of tree skeletons as well as describes cases where our interface design works effectively. This chapter also reports the results of preliminary experiments.

The following are our major contributions:

- We have designed an interface between tree skeletons and tree-structure implementation on the basis of iterators (Section 3.3). We describe the benefits of our design for tree skeleton libraries (Section 3.5).
- We have implemented our interface on the basis of C++ templates and implemented array-based tree skeletons [Mat07a] by using our interface (Section 3.4). Our new implementation tames enormous error messages at compile time by using a type-checking technique.
- We report the results of preliminary experiments on our new implementation of tree skeletons (Section 3.6). No significant overhead of the use of our interface was observed. The flexibility and efficiency of the implementation derived from our interface design were demonstrated.

## 3.2 Tree Skeleton Library

In this section, we introduce the SkeTo library. SkeTo were implemented in C++ with MPI[2] for distributed-memory machines such as clusters. It provides data-parallel skeletons as generic function templates that take function objects and the distributed implementations of their target data structures.

SkeTo is based on the single-program multiple-data (SPMD) model along MPI. A program that uses SkeTo runs in multiple MPI processes. Each process performs the same computation by default except for the internals of skeletons and distributed data structures. For example, we can use list skeletons in SkeTo as shown Figure 3.1. A notable point is that each variable provides the same view among all processes. All parallel computing is closed in the internals of skeletons and distributed data structures. This design is not specific to SkeTo but common in data-parallel skeleton libraries.

An experimental version of SkeTo contained an array-based implementation [Mat07a] of tree skeletons. As mentioned in Section 2.3, segmented trees are well-suited to distributed memory. The distributed implementation of segmented trees is as follows. The entire of each local tree lies in the local memory of a process. Global trees are replicated among all processes because their sizes are reasonably small. Each node of a global tree contains a segment and the rank of a MPI process to which we assign the segment. A global tree lying in the local memory of each process includes only the segments assigned to the process. That is, the replicated global tree in a process is different in locally lying segments from that in another process. In the rest of this chapter, we intentionally confuse MPI processes with processors in parallel algorithms and do MPI process ranks with processor numbers.

Even though the global tree of a segmented tree is replicated among all processors, computations on global trees require collective communications because of intermediate results for segments that do not locally lie. The root processor gathers scattered intermediate results, performs computations on global trees exclusively, and scatters the results over the other processors. Replicated global trees specify the sources and destinations in message passing.

[1] `http://sketo.ipl-lab.org/`
[2] `http://www.mpi-forum.org/`

```
int sketo::main(int argc, char **argv)
{
  // All processes perform the following two lines equally.
  int *raw_array = new int[n];
  initialize(raw_array, n);

  // Construction of a distributed list of length n.
  sketo::dist_list<int> dl(n, raw_array);

  // Application of list skeleton map with a function object f.
  sketo::dist_list<int> ret = sketo::list_skeletons::map(f, dl)

  // Gather distributed data of ret to raw_array.
  // All processes can observe the elements of ret via raw_array.
  ret.gather(raw_array, n);

  // Application of list skeleton reduce with +.
  // All processes can observe its result via sum.
  int sum = sketo::list_skeletons::reduce(std::plus<int>(), 0, dl);

  return 0;
}
```

Figure 3.1: Example use of list skeletons in the SkeTo library.

## 3.3 Our Interface of Trees

In this section, we describe our interface of trees for tree skeletons. The primary part of our interface is an iterator for trees. The concept of iterators itself is very common; they are extensively used in C++. The iterator is only an abstraction of pointers. The requirements of iterators for tree skeletons are important. We describe our iterators in the following two subsections.

### 3.3.1 Iterators over a Tree

Tree structures are necessary for computation on trees. However, an iterator that points to a node does not have to return its children. We have only to be able to perform both bottom-up and top-down reductions of trees by using iterators. For example, we can perform both reductions of a tree in its preorder/postorder traversal by using a stack. Actually, preorder/postorder traversal is essential for depth-first search, and depth-first search is a traversal pattern appropriate to both reductions. We therefore base iterators upon preorder/postorder traversal.

We do not have to extract from a node its children, but we have to know whether it is a leaf node or an internal node. Otherwise, we could not reduce a tree by using a stack. A requirement of preorder/postorder iterators is therefore to return node symbols as well as payload values.

Postorder and reverse preorder are more appropriate for bottom-up reduction. Preorder and reverse postorder are more appropriate for top-down reduction. Which of preorder and postorder is more appropriate relies on input/output formats. XML, which is a popular serialization format of trees, adopts preorder. We therefore adopt preorder and reverse preorder for iterators over a tree.

### 3.3.2 Global Iterators and Local Iterators

A segmented tree is a two-tiered tree; a segment is a local tree and all the segments constitute a global tree. It is therefore natural to define an iterator over a local tree, i.e., a *local iterator* and an iterator over a global tree, i.e., a *global iterator*. Figure 3.2 illustrates the traversals of local/global preorder iterators.

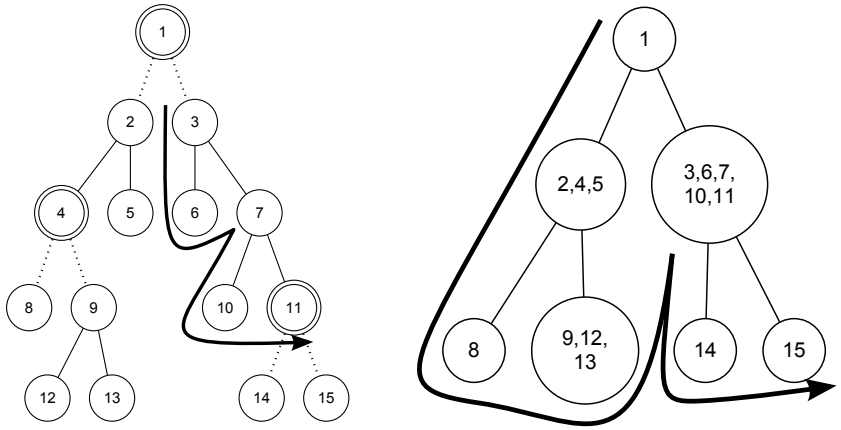



Figure 3.2: Illustration of traversals of local/global preorder iterators. On left is traversal of local iterator of segment $\{3, 6, 7, 10, 11\}$; on right is that of global iterator.

Since tree skeletons require similar computational patterns for both trees, we can use preorder iterators and reverse preorder ones for both local ones and global ones. However, there are different requirements for both iterators.

On the side of local trees, iterators have to take account of hole nodes. Hole nodes are leaves in local trees but are internal nodes in the whole of a segmented tree. The operators to apply to hole nodes therefore can differ form those to apply to true leaf nodes in reducing local trees. A requirement of local iterators is to distinguish hole nodes in addition to leaf ones and internal ones.

On the side of global trees, their nodes have additional information: processor assignment. To enable skeletons to perform an arbitrary collective communication of the distributed data of global trees, the information of processor assignment must be shared among all processors. A requirement of global iterators is therefore to return the processor assignment of a target segment.

There is an additional recommendation for global trees because their sizes dominate the amount of intermediate data. For example, in reduce, each segment is reduced into a single intermediate value. We have to align and buffer such intermediate data in collective communication. The number of the segments assigned to a processor determines this buffer size. Although we can calculate it through iterations by a global iterator, it is redundant to calculate it again on each collective communication. We therefore consider that the size information of a global tree should be cheaply accessible without iterations by global iterators and without communication.

### 3.3.3 Tree Construction

Another important point of our interface is tree construction. In existing implementations of skeletons, calls of the constructors of data structures are hard-coded. This hard coding causes tight coupling between the implementations of skeletons and those of data structures. Skeletons should not determine the implementation of constructors; constructors should be given to skeletons. In our design, the implementation of trees determines the constructors used in skeletons. Tree construction required in tree skeletons is to clone the same shaped tree of a given tree. Therefore, our interface has a shape-cloning constructor of trees. Shape-cloning constructors should be able to construct trees of $SegTree_\alpha$ for any $\alpha$ but their payload values are unspecified.

## 3.4 Our Implementation

We implemented our interface and tree skeletons in the SkeTo library. SkeTo is implemented in C++ with MPI for distributed-memory machines. SkeTo adopts the SPMD model and conceals parallel execution

```
template <typename A>
struct global_tree_impl : factory_impl {
  typedef global_iterator_impl1<A> preorder_iterator;
  typedef global_iterator_impl2<A> reverse_preorder_iterator;
  typedef segmented_tree<A> tree_type;
  preorder_iterator begin();
  reverse_preorder_iterator rbegin();
  int num_segments();
  int num_leaf_segments();
  int num_internal_segments();
  int num_local_leaf_segments(int p);
  int num_local_internal_segments(int p);
  int max_num_local_leaf_segments();
  int max_num_local_internal_segments();
};

struct factory_impl {
  template <typename B>
  struct new_tree {
    typedef global_tree_impl<B> type;
  };
  template <typename A, typename B>
  static void new_shape_clone(const global_tree<A>& src,
                              global_tree_impl<B>*& dst);
};

template <typename A>
struct local_tree_impl {
  typedef local_iterator_impl1<A> preorder_iterator;
  typedef local_iterator_impl2<A> reverse_preorder_iterator;
  preorder_iterator begin();
  reverse_preorder_iterator rbegin();
};
```

Figure 3.3: Simplified implementation of our tree interface.

from users. Each skeleton in SkeTo returns the same value at every process except for the internals of distributed data structures.

In this section, we describe our implementation of tree skeletons. Our implementation of both algorithms and data structures was based upon prior work [Mat07a]. Refer to it for the details.

### 3.4.1 Template-based Implementation of Our Tree Interface

We implemented our tree interface on the basis of C++ templates. Figure 3.3 shows an implementation of our tree interface in C++ templates. The class templates `global_tree_impl` and `local_tree_impl` respectively implement the interface of global trees and that of local trees. Both provide the types of iterators as members `preorder_iterator` and `reverse_preorder_iterator`. A member function `begin()` returns a preorder iterator at the root of a receiver tree. A member function `rbegin()` returns a reverse preorder iterator at the last of the preorder traversal of a receiver tree. Member functions suffixed with `_segments` return size information on a receiver global tree in constant time. Here, leaf/internal segments mean those that are leaf/internal nodes in their global tree; local segments at processor $p$ mean ones assigned to $p$. The numbers of them are not necessarily required for implementing tree skeletons

```
template <typename A>
struct global_iterator_impl {
  node_tag_type tag();
  int proc();
  local_tree<A>& segment();
  global_iterator<A>& operator++();
  bool available();
};

template <typename A>
struct local_iterator_impl {
  node_tag_type tag();
  A& leaf_value();
  A& branch_value();
  A& terminal_value();
  local_iterator<A>& operator++();
  bool available();
};
```

Figure 3.4: Simplified implementation of our iterator interface.

but are often useful for implementing typical communication patterns. Only the `tree_type` member of `global_tree_impl` is part of our interface for type checking rather than our tree interface. Its type `segmented_tree<A>` merely declares $SegTree_{\mathtt{A}}$ by the type name for type checking and therefore tree skeletons do not care about the definition. We shall explain the details later.

We designed the functionality of tree construction as a separate class, which we called a *factory*. `factory_impl` implements a factory of `global_tree_impl`. A factory has two members: a function template `new_shape_clone()` and a class template `new_tree`. The former is an implementation of the construction of shape-cloned trees. The latter works as a type-level function that returns the concrete type of shape-cloned trees constructed in `new_shape_clone()`, given a type of their payload values. For example, given `int`, `new_tree<int>::type` returns `global_tree_impl<int>`, a specific class that implements our tree interface. Both members are also part of the interface of global trees. This is why `global_tree_impl` inherits `factory_impl`.

Figure 3.4 also shows an implementation of our iterator interface. Both local and global iterators support the prefix unary `++` operator overloading and have member functions `tag()` and `available()`. The prefix `++` to iterators makes them take a step forward; `available()` tests whether a receiver iterator has reached the end of traversal; `tag()` returns the tag of a current node. A tag represents node symbols as well as whether a node is the hole. A member function `segment()` of global iterators returns a current segment and `proc()` returns a process rank assigned to the segment. Member functions `leaf_value()`, `branch_value()`, and `terminal_value()` of global iterators return the payload value of a current node. Which of the three we can call is determined by using `tag()`. Although the payload values of a tree are of the same type currently, our iterator interface is designed to support a tree whose payload values are of different types between leaf nodes and internal ones.

Note that tree skeletons require only members that consist of our tree interface including factories and iterators. They do not impose any other requirement on implementation such as the inheritance of a specific class.

For brevity, we omit `const` iterators, whose pointee objects are read-only as pointed by `const` pointers, from Figures 3.3 and 3.4. The `const` iterators over a tree work as input iterators in tree skeletons, while non-`const` iterators work as output iterators.

We implemented tree structures by using the array-based representation given in prior work [Mat07a]. It stores the nodes of a tree into an array in preorder. A main difference from the prior work is that we store each of data members (i.e., tags and payload values) into a separate array. That is, we adopted a

structure-of-array representation in contrast to the array-of-structure one adopted in the prior work. As a result, we saved the space for tags successfully.

### 3.4.2 Generic Implementation of Tree Skeletons

The algorithms that we have implemented are the same as those of prior work [Mat07a]. A traversal of trees was implemented as a loop with use of a stack. We used an input iterator of an input tree for the reduction of trees. We used an input iterator of an input tree and an output iterator of an output tree for the accumulation of trees. In the accumulation, an input iterator and an output one run in lockstep, and an output tree is updated through the output iterator. This output iterator is write-only. We used no iterator for intermediate data because we store them into raw arrays; instead we used indices simply.

When skeletons perform computation over a global tree, they first gather intermediate data distributed among all processes onto the root process. Then, the root process performs computation over a global tree, and finally distributes its resultant data to all other processes. In communication, we have to buffer intermediate data. Our implementation does not serialize payload values but pack them simply into buffers. We implemented the overlap of packing and communication by using double buffering.

We segregated intermediate values on the basis of their types. For example, the intermediate values after the first phase of reduce form a global tree whose internal nodes have payload values of type $\gamma$ and whose leaf nodes have payload values of type $\beta$. We represented this intermediate global tree as two arrays: the one of values of type $\beta$ and the one of values of type $\gamma$. We do not have to store node symbols because they are the same as the ones of input global trees. In traversing a global tree, intermediate values are stored into these two arrays one by one. Since the order of node symbols preserves, two-array representation is consistent. Existing implementations had used the union type of $\beta$ and $\gamma$. We do not adopt this approach because the use of the union type in C++ restricts $\beta$ and $\gamma$ to POD types. Besides, the use of `union` is generally space-inefficient. Our two-array representation of intermediate data is appropriate.

We took a little care of the genericity of implementation. We used template parameters (i.e., type parameters) as generally as possible. For example, we did not use the payload-value type of an input tree explicitly in tree skeletons by passing payload values directly to parameter operators. This enables the implementation of tree skeletons to deal with an input tree that has payload-value types different in internal nodes and leaf nodes.

### 3.4.3 Type Checking of Tree Skeletons

We implemented tree skeletons with function templates as generally as possible. As a result, many template parameters were used in their implementation. Such an implementation is known to generate enormous unreadable error messages at compile time [ME10]. This cause is that the semantics of C++ adopts structural typing for template parameters, whose type constraints are implicitly generated in the definitions of templates. When a template parameter breaks a type constraint generated in the depths of template libraries, C++ compilers therefore will show its generated point as its breaking point together with a long backtrace to report type errors. This is not helpful for library users.

We have dealt with this problem by forcing to generate all of type constraints at the signature of each tree skeleton. For example, we prepared a class template `reduce_signature` that generates the type constraints of reduce, for its implementation `reduce`, as shown in Figure 3.5. Now, `reduce_signature` works as a type-level partial function from the valid types of the parameters of reduce to the result type of reduce. Specifically, Its instantiation such as `reduce_signature<F1,F2,F3,F4,F5,F6,T>` corresponds to type-level function application and the `result_type` member of its instantiation corresponds to the result of application. If types given to `reduce_signature` are invalid as the types of the parameters of reduce, `result_type` cannot be accessed, i.e., this type-level application is invalid. It is therefore partial as a type-level function. We used `reduce_signature` for calculating the result type from the template parameters at the signature of `reduce`. Thus, the signature of `reduce` has been undefined with respect to arguments of invalid types. Type error messages for the calls of `reduce` with arguments of invalid types shall arise only at their call sites successfully.

```
template <class KB, class KL, class Tau, class Phi,
          class Psil, class Psir, class Tree>
struct reduce_signature;

template <typename A, typename B, typename C>
struct reduce_signature<B(B,A,B),
                        B(A),
                        C(A),
                        B(B,C,B),
                        C(C,C,B),
                        C(B,C,C),
                        segmented_tree<A> > {
  typedef B result_type;
};

template <class KB, class KL, class Tau, class Phi,
          class Psil, class Psir, class Tree>
typename reduce_signature<typename KB::function_type,
                          typename KL::function_type,
                          typename Tau::function_type,
                          typename Phi::function_type,
                          typename Psil::function_type,
                          typename Psir::function_type,
                          typename Tree::tree_type
                          >::result_type
reduce(const KB &kB, const KL &kL,
       const Tau &tau, const Phi &phi,
       const Psil &psil, const Psir &psir, const Tree &t)
{
  return reduce_impl(/* omitted */);
}
```

Figure 3.5: Simplified snippet of our implementation of reduce. Partial specialization of class template `reduce_signature` works as declaration of signature of reduce.

We assume operator/tree arguments to have the `function_type`/`tree_type` members that are types declared for type checking. This is an interface for type checking. `reduce_signature` checks the `function_type`/`tree_type` members of operator/tree arguments instead of the types of themselves at the signature of `reduce`. The implementation of skeletons does not check whether the declared types of arguments are compatible with the actual types. The compatibility between both is a responsibility on the side of operators and trees.

We can implement the compatibility checking of `function_type`/`tree_type` in a superclass. For example, the partially specialized definition of the `check` class template in Figure 3.6 checks whether its subclass of type `Impl` has the function signature of `R(A1)` in the constructor, and declares the signature as `function_type`. We can check an operator class by making it inherit `check` like `dup_int` in Figure 3.6. If the signature of `operator()` of `dup_int` were incompatible with `pair<int,int>(int)`, which is declared in the inheritance of `check`, its compilation would fail and a type error message would arise successfully in the constructor of `check` instantiated from the definition of `dup_int`. Note that this type checking is safe regardless of the implementation of `operator()` because its call site in the constructor of `check` is not realizable in runtime.

We have separated the type checking of individual arguments of skeletons from that of application of skeletons, by using the type-checking interface based on the `function_type`/`tree_type` members. Since

```
template <class Impl, typename Signature> struct check;

template <class Impl, typename R, typename A1>
struct check<Impl,R(A1)> {
  typedef R function_type(A1);
  check() {
    if (false) {
      R r = static_cast<const Impl&>(*this)(A1());
    }
  }
};

struct dup_int : public check<dup_int,pair<int,int>(int)> {
  pair<int,int> operator()(int x) const;
} op;
```

Figure 3.6: Example for checking signatures of operators. Superclass template `check` verifies and declares signature of call operation of subclass `dup_int`.

the arguments of skeletons are required to have these members, this style of type checking is intrusive, i.e., opposite to the non-intrusive style conventional in C++. Although we can implement type checking in a non-intrusive style through type-level computation based on the C++11 standard at the signature of skeletons, we have adopted this member-based intrusive style to achieve a clear separation of concerns in type checking. Consequently, we have obtained both a clear declaration of the signatures of skeletons and helpful messages of type errors on individual arguments. These benefits from separation of concerns in type checking is particularly valuable for tree skeletons because their signatures are complicated.

Note that we do not claim the novelty of our type-checking technique. Type checking of template parameters is a common issue in C++ template metaprogramming. In fact, the C++ concept [SS12], which is a functionality for general type checking of template parameters, is under discussion on the next C++ standard and partially emulated by the Boost Concept Check Library[3]. Rather than a general and/or versatile one, our type-checking technique is a minimalism specialized for checking function signatures. For example, we can sophisticate the type-checking code of `check` in Figure 3.6 by using type traits, which are type-level predicates, and `static_assert` in the C++11 standard. Nevertheless, our implementation is simple yet sufficient to generate helpful error messages even in the C++03 standard.

## 3.5 Benefits of Our Design

Our interface design of tree skeletons is greatly advantageous to their implementation as well as has great potential to improve data-parallel skeleton libraries. In this section, we describe cases where our design works efficaciously.

Our design is particularly valuable for tree skeletons owing to the heterogeneousness and versatility of trees. The heterogeneousness of trees strongly motivates skeleton implementors to specialize representations of trees. The versatility of trees strongly motivates skeleton users to switch views of trees. We explain these in the following subsections.

### 3.5.1 Specialization of Representation

Trees are heterogeneous. For example, the types of payload values may not be uniform. Consider (untyped) nested lists. A nested list is represented by a tree whose leaf nodes are the elements of the nested list and whose internal nodes have no payload value. Such trees are not unusual. Generally, a tree

[3] `http://www.boost.org/doc/libs/release/libs/concept_check/`

that represents a recursive decomposition of a set has meaningful data only in its leaf nodes. Such trees can be found in intermediate results. The result of path-wise reduction (i.e., a reduction version of dAcc) is directly represented by a tree that has no internal payload value. For efficiency of parallel computing, the representation of such trees are desired to be specialized to save the space of their internal nodes.

Not only in payload values, trees may be also heterogeneous in structures. For example, binary trees may be non-full, i.e., have a single-child internal node with a missing leaf, while tree skeletons accept only full binary trees for the brevity of these APIs. A way to deal with non-full binary trees in tree skeletons is, as mentioned in Section 3.1, to fill these missing leaves with dummy nodes of a default value. For efficiency, such filled trees are desired to have a specialized representation to save space of dummy nodes. Besides, we may have to deal with trees with high-degree nodes (i.e., hubs). We can also deal with such a tree by converting the whole tree into a full binary tree. For space efficiency, however, it is natural to give hubs special treatment in representation.

The input/output formats of trees used in practice are also heterogeneous. For example, both XML and JSON are widely used in the web. In relational databases, nested sets [Cel12] are popular. For efficiency, it is valuable to specialize the representation of trees in a specific format. Actually, the preservation of the bracketed structures of XML documents leads to an efficient implementation of tree skeletons [KME07].

A more advanced case is implementation of template-based fusion methods [ME10, EM14]. These are a technique to eliminate intermediate data by decorating data structures with specific computations. Because decorated data structures pretend the results of the computation of decoration, they avoid constructing the whole of the results. This technique can be seen as a specialization of representations equipped with computations.

In summary, skeleton implementors are greatly motivated to specialize the representations of trees from the perspective of efficiency. If the implementation of tree skeletons is tightly coupled with that of trees, we have to implement tree skeletons for each implementation of trees. This is excruciatingly unproductive. The implementation of tree skeletons loosely coupled with that of trees enables us to avoid such unproductiveness. Our design achieves such a loosely coupled implementation of skeletons and therefore promotes specialization of tree representation.

### 3.5.2 Multiple Views

Trees are versatile. We can find uses of trees for modeling something in various contexts. This means that a tree probably has another view of data. For example, consider XML documents. They are usually modeled as DOM trees but are actually bracketed texts. We would sometimes like to deal with them as texts regardless of their brackets. For example, consider `grep` (i.e., regular expression matching and filtering) for texts that have dropped their brackets. We can implement `grep` for texts by using prefix sums [Ble93], i.e., one of list skeletons. We would then like to use a list view of a DOM tree.

Of course, in this case, it is sufficient to flatten a DOM tree into a list of its leaf texts. However, data-parallel skeleton libraries hardly deal with such flattening because construction of data structures is generally restricted in parallel computing. We can elude such flattening by encoding `grep` on trees but this is not desirable for skeleton users. An implementation of trees that provides a list view of leaves is appropriate both to implementation and practical use.

Our design is appropriate for providing multiple views. Our interface defines a view of trees. Whether its underlying implementation is a tree or not, an implementation of our interface behaves as a segmented tree for tree skeletons.

An inverse of this is also possible. For example, we consider a way of extending our implementation of binary trees to provide a list view of its leaves. An important point here is that a one-hole context corresponds to two local lists of leaves. A simple way to dealing with it is to assign indices of local lists to each local tree. We assign an index to a subtree and two indices to a one-hole context. Then, we define a global iterator for the list view as an unordered iterator of indexed elements. It traverses a global tree and respectively returns the local lists in each segment with their indices. We would hardly change a local iterator. It is sufficient to skip internal nodes. Since two local lists are separated at the hole node in a one-hole context, a local iterator over a one-hole context begins its root or its hole. We consider in general that the design of global iterators shall be a key.

Table 3.1: Execution time (in seconds) of tree skeletons with/without our interface.

| No. of nodes, $N$ | With our interface | | | Without our interface | | |
|---|---|---|---|---|---|---|
| | reduce | uAcc | dAcc | reduce | uAcc | dAcc |
| $128 \times 2^{20}$ | 0.753 | 1.360 | 1.127 | 0.658 | 1.425 | 1.285 |
| $256 \times 2^{20}$ | 1.518 | 2.677 | 2.250 | 1.332 | 2.780 | 2.509 |
| $512 \times 2^{20}$ | 3.053 | 5.471 | 4.538 | 2.659 | 5.666 | 5.051 |

To provide multiple views is also valuable for efficient implementation. Existing tree skeletons necessitate the preprocessing of input trees of non-full binary ones or unbounded-degree ones. This preprocessing is large overhead. Without actual preprocessing, it is sufficient to provide a view of preprocessed trees. In particular, we should not encode unbounded-degree trees into full binary trees. Parallel tree contraction, which is the basis of tree skeletons, can be applied to unbounded-degree trees more efficiently than binary trees [MM11a]. Tree skeletons should therefore provide a way to operate unbounded-degree trees as they are. Our interface is useful for implementing trees compatible both with binary tree skeletons and unbounded-degree tree ones.

In summary, our design leads to multiple views of trees as well as various data structures. Multiple views are particularly valuable for trees. They enable us to use a combination of skeletons over different data structures although this is difficult to do in existing data-parallel skeleton libraries. Our design therefore has great potential to improve data-parallel skeleton libraries in general.

## 3.6 Preliminary Experiments

We conducted preliminary experiments on our new implementation of tree skeletons. In this section, we report these results.

### 3.6.1 Overhead of Our Interface

To evaluate the overhead of the use of our interface, we measured the execution time of tree skeletons with/without our interface. We used the existing implementation [Mat07a] as tree skeletons without our interface. Parameter operators were addition over `int` and the identity function, i.e., sufficiently cheap to measure the cost of traversals.

Our new implementation differs a little from the existing one, particularly on communication. This difference is independent of the use of our interface. To minimize the effect of communication, we ran two processes[4] on a single shared-memory server: one equipped with four Opteron 6380 (16 cores, 2.50 GHz) processors and 128 GB of DDR3-1600 memory running Debian 7.5 (Linux 3.2.0-4-amd64). We compiled source code by using g++ 4.7.2 with the O3 optimization, together with OpenMPI 1.4.5. This setting in principle does not work beneficially to our new implementation.

Input trees were randomly generated ones whose payload values were of `int`, their every segment had the same size artificially for minimizing the effect of load imbalance, and the number of segments was 65. We measured the average time of 10 times trials for the same input. Table 3.1 summarizes the results of this experiment.

Slight slowdowns were observed in reduce and slight speedups were observed in uAcc and dAcc at every input size. We consider these differences to be insignificant. If the use of iterators incurred some overhead, a uniform slowdown in all skeletons would be observed. We attribute both speedup and slowdown to extrinsic factors. For example, to avoid using `union`, our new implementation adopts a two-array representation. As a result, MPI Isend/Irecv calls in reduce and uAcc increase although the amount of communication data does not increase. This might have incurred slowdown. In contrast, our new implementation of trees saves the space of tags. This might have brought speedup. We consider these extrinsic factors to be insignificant. In conclusion, we observed no significant overhead of the use of our interface.

[4]Neither MPI-based implementation was able to run with a single process.

```
// initializations
dist_tree<A>         t = ...;
dist_leaf_tree<A>  lt = ...;
// simple applications
dist_tree<B>        t2 = dAcc(..., t);
dist_leaf_tree<B> lt2 = dAcc(..., lt);
// preparations for conversion
factory_hijacker<dist_tree<A>,dist_leaf_tree_factory> wt(t);
factory_hijacker<dist_leaf_tree<A>,dist_tree_factory> wlt(lt);
// converting applications
dist_leaf_tree<C> lt3 = dAcc(..., wt);
dist_tree<C>        t3 = dAcc(..., wlt);
```

Figure 3.7: Simplified example of simple application and converting application of skeleton.

### 3.6.2 Flexibility from Our Interface

We demonstrate the extensibility and flexibility of implementation based on our interface through experimental implementations. Our basic implementation of trees mentioned in Section 3.4 was `dist_tree<A>`. In addition, we developed two implementations of trees. One was the one of trees whose internal nodes have no payload value, `dist_leaf_tree<A>`, of which we saved the space of the internal nodes (see Section 3.5.1). The other was a wrapper of trees, `factory_hijacker<T,F>`. It behaves as a tree `T` whose factory is overwritten by `F`. By using this wrapper together with factories of `dist_tree<A>` and of `dist_leaf_tree<A>`, we can switch the implementations of the output trees of tree skeletons from the caller side, as shown in Figure 3.7.

The example program, whose text is slightly simplified, shown in Figure 3.7 has two notable points. The first is that dAcc had the only implementation regardless of the differences on implementation both of input trees and output ones. This demonstrates the genericity of our implementation of tree skeletons. Note that our implementation of skeletons is independent of the implementation of output trees unlike conventional functions of parametric polymorphism and does not necessitate upcasting the types of output trees unlike conventional functions of class-based polymorphism. In this sense, our implementation is generic yet more flexible than conventional ones. This flexibility is due to template-based implementation of our tree interface.

The second is that our implementation of skeletons enables converting applications. In simple applications, the implementations both of input trees and output ones were the same. In converting applications, the implementations of output trees were converted from those of input trees through `factory_hijacker<T,F>` in dAcc. Note that converting applications are not different from simple applications from the perspective of implementation. `factory_hijacker<T,F>` itself only attached factories without modifying trees. Each factory only called a specific constructor and initialized a tree. dAcc only wrote an output tree in the same way. Even though all these were unaware of conversion, the implementations of trees were yet successfully converted. This exemplifies separation of concerns and also demonstrates the extensibility and flexibility of implementation based on our interface.

### 3.6.3 Avoidance of Data Restructuring

To demonstrate the feasibility and benefits of multiple views, we implemented a list interface into `dist_leaf_tree<A>` as described in Section 3.5.2. We experimentally implemented three kinds of reduce for the leaf list of `dist_leaf_tree<A>`.

The first was a hand-coded implementation of leaf-list reduction. The computation over a local tree utilized the implementation of `dist_leaf_tree<A>` in a tightly coupled manner and the computation over a global tree was similar to that in reduce of tree skeletons but specialized for leaf-list reduction. It was a baseline with no overhead. We name it the *hand-code* version.

Table 3.2: Execution time (in seconds) of leaf-list reduction given $256 \times 2^{20}$ leaves.

| | No. of processes, $P$ | | | |
|---|---|---|---|---|
| | 2 | 3 | 6 | 9 |
| hand-code | 0.560 | 0.436 | 0.238 | 0.149 |
| list-view | 0.688 | 0.544 | 0.309 | 0.195 |
| raw-list | 14.989 | 18.706 | 19.035 | 19.162 |

The second was a generic iterator-based implementation of reduce of list skeletons. `dist_leaf_tree<A>` constructed a list view of leaves internally and exposed it to the generic implementation. This achieved loose coupling between the implementation of `dist_leaf_tree<A>` and that of reduce of list skeletons. We name it the *list-view* version.

The third was a reference implementation of list skeletons tightly coupled with an implementation of distributed lists. This distributed list was constructed from a single array in the root process by scattering its data. We implemented, into `dist_leaf_tree<A>`, a gather operation that aligns the payload values of all leaves to a single array on the root process. The reduction of all leaves of `dist_leaf_tree<A>` is the gather operation followed by the reference reduce. We name it the *raw-list* version.

We compared the execution time of these three implementations. The aim of this experiment is to measure the overhead of data restructuring between lists and trees. Parameter operators were therefore the same as ones used for measuring the overhead of our interface. We used as input, the tree of $512 \times 2^{20}$ nodes that was randomly generated before. It had about $256 \times 2^{20}$ leaves.

Communication much concerns the overhead of data restructuring. To measure realistic overhead on distributed memory, we used a three-node cluster each of whose node was equipped with two Opteron 2376 (4 cores, 2.3 GHz) processors and 8 GB of DDR2-800 memory running Ubuntu 12.04.4 (Linux 3.2.0-64-generic) and was connected each other through 1000BASE-T. We compiled source code by using g++ 4.6.3 with the O3 optimization, together with OpenMPI 1.4.3. Processes were evenly assigned to each node.

Table 3.2 summarizes the result of this experiment. raw-list was much slower than both hand-code and list-view, and showed no scalability against the number of processes $P$. This cause was communication. The gather operation used in raw-list costs the amount of communication data proportional to $P$, while hand-code and list-view cost no extra communication to perform reduce. The construction of a list view in list-view incurred slight overhead but did not spoil scalability owing to no communication. Furthermore, since this list view shares the underlying data of `dist_leaf_tree<A>`, we do not have to reconstruct the list view as long as tree structures are unchanged. In practice, the overhead of view construction would therefore be more negligible.

In summary, the use of a list view of trees brought significant performance gain owing to the avoidance of data restructuring as well as a generic implementation of skeletons. Multiple views that our interface design can provide are therefore very beneficial to efficient implementations of skeletons and will promote a crossover use of skeletons over different data structures.

## 3.7 Related Work

In the context of functional programming, where trees are extensively used in the form of algebraic data types, abstractions for trees were well studied. For example, active patterns [SNM07] generally enable us to perform pattern matching to abstract data types as algebraic ones; i.e., it provides a tree interface of abstract data types in a general manner. Most of such studies did not deal with parallel programming. A notable exception was an abstraction based on ternary trees [MM11b]. It formalized the algebraic structure of parallel tree contraction as the view of an algebraic data type of ternary trees. On these ternary trees, load balancing forms as tree balancing. Another notable work was a formalization of path-based computations on trees [MMHT09]. Its list-like view led to the operations of parallel tree contraction.

In the context of parallel programming, computation on nested lists, which are a view of trees as

mentioned in Section 3.5, is popular. Flattening [Ble90] of nested lists is a fundamental technique for load balancing. To elude large space overhead of a thorough flattening, Bergstrom et al. [BFR+13] developed a hybrid flattening that provided a flat view and a nested one. This is an application of trees with multiple views mentioned in Section 3.5.2.

In skeleton libraries, Triolet [RJDH14] was relevant to our work. It utilized hybrid iterators that provide both a nested list view and a flattened list view for a better fusion strategy of a filter skeleton. This notion is similar to the hybrid flattening above. Triolet used hybrid iterators only for list skeletons. Our iterators are designed for tree skeletons and our interface design aims at even a crossover use of skeletons over different data structures.

Iterators are commonly used in skeleton libraries. For example, Java 8 collections[5] used Spliterator and parallel collections in Scala 1.9 [PBRO11] used IterableSplitter. These abstracted a recursive splitting of collections but did not concern the structures of parallel collections. That is, these were not designed for structural computation on parallel collections. STAPL Parallel Container Framework (PCF) [TBF+11], which was inspired by C++ STL, also used iterators called pView. STAPL pView classified accesses to parallel data structures. STAPL PCF shared an objective with our library design in the sense that it strove after both parallel computing and loose coupling like C++ STL. Although STAPL PCF could address a structural computation to some extent, through tree skeletons, we tackle with more general and/or complicated structural computations. Our interface design aims at such structural computations on various parallel data structures.

Several parallel graph libraries for distributed-memory machines such as the Parallel Boost Graph Library[6] [GL05] and the STAPL parallel graph library [HFAR13] were analogous to C++ STL. Since our interface design is also inspired by C++ STL and the tree is of course a graph, these were relevant to our work. While our tree interface is based on depth-first search, these provided breadth-first search visitors. While our interface is designed for library implementors, these visitors were designed for library users. We deal with coarse-grained parallelism in structural computation on trees in order to minimize communication and synchronization.

## 3.8 Conclusion

We have presented the design and implementation of tree skeletons based upon our iterator-based interface of trees. We implemented our interface appropriately on the basis of C++ templates. Our interface provides loose coupling between the implementation of tree skeletons and that of trees. This is helpful both for skeleton implementors and users.

A direction in future work is to extend our library for providing the multiple views of trees and other data structures. Our extended library would be consist of three tiers. The first is implementation of data structures that satisfy interfaces. The second is implementation of interfaces that define the views of data structures and their cost models like C++ STL. The third is implementation of skeletons based on interfaces. A flexible combination of instances in respective tiers will be a desirable implementation of data-parallel skeletons.

[5] `http://docs.oracle.com/javase/8/docs/api/java/util/Collection.html`
[6] `http://www.osl.iu.edu/research/pbgl/documentation/`

# Chapter 4

# Tree Skeleton Hiding

This chapter is a revised and extended version of our publication [SM13].

## 4.1 Introduction

Tree skeletons [Ski96, GCS94] are indeed high-level abstractions for parallel programming. We can use tree skeletons with no concern for low-level details of parallelism and underlying parallel machines. In this sense, parallel programs based on tree skeletons are portable. Once we describe parallel programs with tree skeletons, they can run in various environments on the basis of appropriate implementations of tree skeletons. Moreover, tree skeletons guarantee their asymptotic performance regardless of the shape of trees. In this sense, parallel programs based on tree skeletons are robust. They can run effectively in parallel for various shapes of input. The implementations [SM14a, Mat07a, KME07, EI12, MM14] of tree skeletons certainly provide high-level abstractions potentially equipped with these virtues. OK then, is it easy to use tree skeletons? No, it is unfortunately not.

The API of tree skeletons is high-level yet complicated. This is the main reason why it is not easy to use tree skeletons. In particular, additional operators that the parallel definitions of tree skeletons use for interpreting tree contraction operations impose complicated algebraic requirements on programmers. To predefine these operators separately is unhelpful for programmers because they have semantic dependencies to operators that programmers would like to provide. For example, to use `reduce`, it is reasonable for programmers to define its parameter operators $k_B$ and $k_L$ because both are required in its sequential definition. Then, its additional operators $\tau$, $\phi$, $\psi_l$, and $\psi_r$ have semantic dependencies to $k_B$ and $k_L$, and cannot be separately defined. Without any assistance, programmers have to derive appropriate definitions of the additional operators. Tree skeletons thus impose a heavy burden on programmers.

Although these additional operators are a key to guaranteeing excellent load balancing of tree skeletons, they are unnecessary from the perspective of sequential specifications. In this sense, they are *auxiliary* operators for load balancing, while operators required in the sequential definitions are *primary* operators for specification. What programmers would like to specify are primary operators. The burden of auxiliary operators spoils the usability of tree skeletons and even outweighs their benefits from the viewpoint of programmers.

To resolve this problem, we have implemented a parallelizer that transforms a recursive function described in a restricted C languages into the application of a tree skeleton by generating operators on the basis of the formalization by Matsuzaki et al. [MHT06]. Our parallelizer enables programmers to use tree skeletons implicitly. As a result, programmers can enjoy the benefits of tree skeletons with no burden. In fact, that our parallelizer hides tree skeletons from programmers is more than the freedom from operator derivation. Our parallelizer gives discretion to design the APIs of tree skeletons regardless of the viewpoint of programmers. As a consequence, more expressive but potentially complicated APIs can be adopted into tree skeletons, and then programmers can also enjoy this expressive power implicitly. Moreover, our parallelizer enables programmers to test a target function simply as a sequential C program because it deals with a restricted C language extended with compiler directives. By abstracting and

hiding tree skeletons themselves, our parallelizer brings many benefits.

The following are our main contributions:

- We have developed a parallelizer for recursive functions described in a restricted C, on the basis of the formalization by Matsuzaki et al. [MHT06]. Our parallelizer hides tree skeletons and provides useful declarations for parallelization as compiler directives (Section 4.3). It has been integrated with a C compiler. We also report the results of a preliminary experiment (Section 4.5).
- We present an approach based on an implicit use of tree skeletons to parallel programming on trees through our parallelizer (Section 4.6). Our approach enables programmers to use more general but complicated tree skeletons (Section 4.4) with no burden and therefore will be generally advantageous to programming based on tree skeletons.

## 4.2 Deriving Operators from Multilinear Computation

In this section, we briefly describe the formalization of systematic derivation of auxiliary operators for tree skeletons. Refer to [MHT06, Mat07b] for a more formal description in the case of full binary trees.

### 4.2.1 Multilinear Computation on Trees

We first formalize target computations on trees. Let $child_i(t)$ be the $i$th-child subtree of a tree $t$. For the sake of clarity, we also use $l$ and $r$ for the suffix $i$ instead of 1 and 2 if $t$ is a binary tree. Letting $f$ be a recursive function on trees that yields a vector, $f$ is a *multilinear* function if $f$ can be defined as

$$f(t) = A_i f(child_i(t)) \quad \text{for all } i. \tag{4.1}$$

Note that $A_i$ is a coefficient matrix that may contain components of the result of $f(child_j(t))$, where $j \neq i$. The leaf case in the above is unspecified because the domain of $i$ is empty. Intuitively, our target is a bottom-up computation that consists of linear transformations of the result for each subtree.

For example, the following *poly* over $BinTree_\alpha$ is multilinear.

$$\begin{aligned} &poly(Leaf(x)) = (x, 1),\\ &poly(Branch(x, l, r)) = (xy_{l2}y_{r2}, (y_{l1} + y_{l2})(y_{r1} + y_{r2})),\\ &\quad \text{where } (y_{l1}, y_{l2}) = poly(l),\\ &\qquad\quad (y_{r1}, y_{r2}) = poly(r). \end{aligned}$$

This is because we can transform the case of $Branch(x, l, r)$ into

$$poly(Branch(x, l, r)) = \begin{pmatrix} 0 & xy_{r2} \\ y_{r1} + y_{r2} & y_{r1} + y_{r2} \end{pmatrix} \begin{pmatrix} y_{l1} \\ y_{l2} \end{pmatrix} = \begin{pmatrix} 0 & xy_{l2} \\ y_{l1} + y_{l2} & y_{l1} + y_{l2} \end{pmatrix} \begin{pmatrix} y_{r1} \\ y_{r2} \end{pmatrix}.$$

The above equations satisfy the form of the equation (4.1).

The equation (4.1) seemingly does not cover the following *multi-affine* function:

$$f(t) = A_i f(child_i(t)) + \mathbf{b}_i \quad \text{for all } i.$$

The constant term vector $\mathbf{b}_i$ may similarly contain components of the result of $f(child_j(t))$, where $j \neq i$. By using an augmented matrix and vector, we, however, can represent the equation above as

$$\begin{pmatrix} f(t) \\ 1 \end{pmatrix} = \begin{pmatrix} A_i & \mathbf{b}_i \\ \mathbf{0} & 1 \end{pmatrix} \begin{pmatrix} f(child_i(t)) \\ 1 \end{pmatrix} \quad \text{for all } i. \tag{4.2}$$

This representation enables us to regard any multi-affine function as a multilinear function. We therefore call target computations *multilinear computations* on trees.

### 4.2.2 Auxiliary Operators in Multilinear Computation

We can consider a multilinear computation for internal nodes to consist of two separate parts: matrix construction and matrix multiplication. In fact, both correspond to the RAKE operation and the COMPRESS operation.

Recall the reduce computation. Only with primary operators, we can reduce a one-hole context to a root-to-hole spine. In multilinear computations, we can construct a matrix for each internal node (except for the hole) of a root-to-hole spine. As a result, the root-to-hole spine reduces to a root-to-hole path (i.e., a list of nodes). This corresponds to the result that we apply RAKE operations to all leaves (except for the hole) of the root-to-hole spine. Then, we can reduce the root-to-hole path to a single node by using matrix-matrix multiplication. Obviously, matrix-matrix multiplication corresponds to the COMPRESS operation.

Since matrix construction and matrix multiplication respectively correspond to RAKE and COMPRESS, by composing both, we can derive auxiliary operators corresponding to the SHUNT operation. However, to formulate auxiliary operators of reduce, we require consideration of the payload values of hole nodes.

The payload value of a hole node is necessary to calculate the result for the subtree rooted by the hole. If we reduce the root-to-hole path to a hole node with matrix multiplication, we lose its payload value. We therefore have to leave payload values after matrix construction. Specifically, we consider an extended payload value that consists of a payload value and a matrix, which is lifted by an auxiliary operator $\tau$. In the matrix multiplication of a parent and child, the payload value part of the extended payload value of the child is left for a later bottom-up computation.

The matrix part of the extended payload value of an internal node summarizes the linear computations on its ancestors. This summary is applied to the result for the internal node. The initial value of the matrix part of each extended payload value is therefore an identity matrix $I$. In summary, we obtain the auxiliary operators of reduce over $\mathit{SegTree}_\alpha$ in the following general form:

$$\begin{aligned}
\tau(x) &= (x, I),\\
\psi_l((x', M'), (x, M), \mathbf{y}_r) &= (x', MA_lM'),\\
\psi_r(\mathbf{y}_l, (x, M), (x', M')) &= (x', MA_rM'),\\
\phi(\mathbf{y}_l, (x, M), \mathbf{y}_r) &= Mk_B(\mathbf{x}_l, x, \mathbf{x}_r),
\end{aligned}$$

where $x$, $\mathbf{y}_l$, and $\mathbf{y}_r$ are implicitly used in $A_r$ and $A_l$.

Note that this formalization itself is applicable to multilinear functions on $k$-ary trees. We can obtain similar general forms of the auxiliary operators of reduce, uAcc, and dAcc over $k$-ary trees.

**Generalization** This derivation of auxiliary operators is based only on the equation (4.1) and the associativity of matrix multiplication. The equations (4.1) and (4.2) necessitate a commutative semiring for the domain of the components of matrices and vectors. This derivation can be therefore generalized for any commutative semiring.

An algebra $(S, +, \times, 0, 1)$ is a commutative semiring if the following axioms of algebra are satisfied.

- The addition $+$ and the multiplication $\times$ have both associativity (i.e., $a * (b * c) = (a * b) * c$ for any $a$, $b$, and $c$) and commutativity (i.e., $a * b = b * a$ for any $a$ and $b$).
- The multiplication $\times$ is distributive over the addition $+$; i.e., $a \times (b + c) = a \times b + a \times c$ and $(a + b) \times c = a \times c + b \times c$ for any $a$, $b$, and $c \in S$.
- 0 is the identity of $+$ and 1 is that of $\times$; i.e., $0 + a = a + 0 = a$ and $1 \times a = a \times 1 = a$ for any $a \in S$.
- 0 is the absorbing element with respect to $\times$; i.e., $0 \times a = a \times 0 = 0$.

The ring of integers $(\mathbb{Z}, +, \cdot, 0, 1)$, the semiring of natural numbers $(\mathbb{N}, +, \cdot, 0, 1)$, tropical semirings $(\mathbb{Z}, \max, +, -\infty, 0)$ and $(\mathbb{Z}, \min, +, \infty, 0)$, and the Boolean semiring $(\{0, 1\}, \vee, \wedge, 0, 1)$ are popular examples found in computer programs. For multilinear computations with all these algebras, we can define auxiliary operators uniformly in the form above.

## 4.3 Proposed Parallelizer

In this section, we describe the design and implementation of our parallelizer.

### 4.3.1 Fundamental Design

#### Aims of Implementation

The fundamental aims of our parallelizer are the following two:

- Hiding tree skeletons from programmers.
- Algorithm descriptions in a conventional style of C.

The main benefit of the first is that programmers become oblivious of complicated APIs of tree skeletons. Tree skeletons implemented in C++ work as a background service. In contrast to skeletons, we do not hide parallel implementations of data structures. It is appropriate that programmers explicitly select and use parallel implementations of data structures because an efficient parallel I/O at construction of data structures often requires some additional consideration for hardware. Our parallelizer therefore generates a function that takes a tree data structure for underlying tree skeletons from each target of parallelization. We call it an *entry function*. An entry function encapsulates tree skeletons as well as operators for them and pretends a sequential function that yields the same result as its source function except for the implementation of argument tree data structures. Our parallelizer therefore enables programmers to focus only on the construction of tree data structures.

The main benefits of the second are ease of programming and the interoperability with C/C++ code. Not all non-expert programmers that desire parallel programs would like functional programming based on grammars. Many textbooks on algorithms still adopt imperative languages for describing algorithms and C-like pointers for representing trees. A restricted C is adequate to describe algorithms on trees and easy for non-expert programmers.

Thee interoperability of algorithm descriptions with C/C++ is also important. We adopt C++ for the language of generated programs because tree skeletons have been implemented in C++. This is not only a historical reason. C++ has the best balance among the expressiveness for high-level abstractions, the portability for parallel machines, and the performance of compiled native code. For the same reasons, we suppose that programs calling entry functions are also described in C++. From the perspective of C++ programming, it is of importance that existing and/or standard C/C++ functions can be used in algorithm descriptions. Although not all C++ functions will be callable for a syntactic reason, a restricted C enables us to use many C/C++ functions in algorithm descriptions.

#### Design for Implementation

To achieve the aims mentioned above, specifically, we have adopted the following design choices:

- Our parallelizer is a C-to-C++ compiler that generates entry functions encapsulated for `#include`.
- The API for parallelization is based on compiler directives (i.e., `#pragma`).
- Targets of parallelization are syntactically restricted recursive functions on a pointer structure.
- Partial definitions of recursive functions over general (i.e., non-full) binary trees are considered.
- Our parallelizer deals with the computation of reduce.

The first is the fundamental design of our parallelizer. For encapsulation of tree skeletons as entry functions, we also have to generate intermediate data types and operators. It is adequate that all generated components are aggregated into a separate include file. The second enables programmers to test algorithm descriptions as sequential C programs by using off-the-shelf C compilers because standard C compilers ignore extensions to compiler directives. The third is the primary design on algorithm descriptions. As found in loop parallelization, certain syntactic restrictions on targets are reasonable

```
#include <stddef.h>
#pragma commSemiring max "+" NINF 0
extern int NINF;
extern int max(int a, int b);

typedef struct BNint {
  int v;
  struct BNint *l, *r;
} BNint;
typedef struct pair_int {
  int _1, _2;
} pair_int;

pair_int mis(BNint *t) {
  pair_int ret, l = {NINF,0}, r = {NINF,0};
  if (t->l != NULL) l = mis(t->l);
  if (t->r != NULL) r = mis(t->r);
  ret._1 = t->v + l._2 + r._2;
  ret._2 = max(l._1, l._2) + max(r._1, r._2);
  return ret;
}
```

Figure 4.1: Example of algorithm descriptions. `mis` calculates the maximum independent sum of a given tree.

for automatic parallelization. The fourth is relevant to handling pointers in a conventional style of C. Algorithm descriptions for trees represented with C-like pointers lead generally to recursive functions over general binary trees. It is natural in C that recursive functions over full binary trees are regarded as partial definitions. We therefore use skeletons over general binary trees (see Section 4.4) and deal with partial definitions of recursive functions. The last is relevant to simplification of implementation. As mentioned in Section 4.2, operator derivation for uAcc and dAcc is straightforward.

#### Example Use

Figures 4.1 and 4.2 show a concrete example of using our parallelizer. An input to our parallelizer is a program that C compilers can compile separately, as shown in Figure 4.1. A generated program (e.g., `mis-tree.cpp`) by our parallelizer is used through `#include`, as shown in Figure 4.2. A generated entry function has the same name as its source function (e.g., `mis` in Figure 4.1) but contained in the `parallel_dp` namespace to avoid name collision. The type `BNint` represents trees of pointer structures for sequential computation and the type `binary_tree<int>` represents trees for an underlying reduce. By giving data of type `binary_tree<int>` instead of `BNint`, we can use the entry function `parallel_dp::mis` as the source function `mis`. A declaration for helping parallelization is a compiler directive such as `#pragma commSemiring` ..., whose meaning is described later.

### 4.3.2 Compiler Directives for Parallelization

Our parallelizer provides three kinds of declarations for helping parallelization in the form of `#pragma`.

#### Declaration of Commutative Semirings

The most important API in our parallelizer is the declaration of commutative semirings. As mentioned in Section 4.2, the formalization of operator derivation is generalized for any commutative semiring. Our parallelizer therefore generates the operators for reduce on the basis of commutative semirings declared.

```
#include <skel/binary_tree.hpp>
#include <skel/config_main.hpp>
#include "mis-tree.cpp"

int main(int argc, char **argv)
{
  config::init(argc, argv);

  binary_tree<int> *t =
    binary_tree<int>::read_from_file("data");
  pair_int result = parallel_dp::mis(*t);

  config::finalize();
  return 0;
}
```

Figure 4.2: Example of using generated code. `mis-tree.cpp` is an output file that our parallelizer generates from the program shown in Figure 4.1.

By using the syntax of `#pragma commSemiring plus times zero one`, we can declare a commutative semiring $(S, \texttt{plus}, \texttt{times}, \texttt{zero}, \texttt{one})$. The domain $S$ is not explicitly declared because we can find the domains (i.e., types) of operators in the context of their use. Our parallelizer thus can easily cope with the overloading of built-in operators and the type conversion of identity symbols.

To use built-in operators for operators of commutative semirings, we have to describe it as a string literal, as shown in Figure 4.1. The commutative semirings of `#pragma commSemiring "+" "*" 0 1` and `#pragma commSemiring "|" "&" 0 1` are implicitly predefined as built-in ones.

We do not check whether commutative semirings declared satisfy the axioms of algebra. This is because the standard representation of numbers and arithmetic in computers do not implement the axioms of algebra strictly. For example, arithmetic overflow and rounding errors make it difficult to implement the axioms of algebra strictly. We therefore assume *approximate* axioms of algebra. The meaning of approximate axioms is, for example, that we use sufficiently small values in calculation process as the absorbing element $-\infty$.

### Declaration of Pure Functions

Coefficient matrices in multilinear computations can contain constants. To enrich the variety of constants, pure functions are useful because the applications of pure (i.e., side-effect-free) functions to payload values and invariant symbols are also regarded as constants. In particular, predicates on payload values such as `all` and `any` in Figure 4.4 are almost essential in multilinear computations with the Boolean semiring. Our parallelizer therefore provides the API to declare pure functions. The syntax of `#pragma pure f1 f2 ...` declares that the symbols `f1`, `f2`, ... are pure functions.

We do not check the purity of declared functions for two reasons. The first is that we assume function definitions to be unknown, i.e., external functions. The declaration of pure functions is provided for reuse of existing functions. If our parallelizer requires programmers to include existing functions in a compile unit, their reusability is spoiled. The second is that it is difficult to define side effects reasonably. Although side effects generally mean the effects of functions except for return values, this definition is impractical in C/C++, where we can observe bare runtime environments. For example, in this definition, math functions such as `sin` have side effects because of `errno`. Moreover, it is controversial whether logging independent of main calculations has to be considered as a side effect. Our parallelizer therefore delegates the decision of pure functions to programmers.

```
pair_int singleton(BNint *t) {
  pair_int ret, l = {0,0}, r = {0,0};
  if (t->l != NULL && t->r == NULL) {
    l = singleton(t->l);
    r._2 = 1;
  }
  if (t->l == NULL && t->r != NULL) {
    l._1 = 1;
    r = singleton(t->r);
  }
  if (t->l != NULL && t->r != NULL) {
    l = singleton(t->l);
    r = singleton(t->r);
  }
  ret._1 = l._1 + r._1;
  ret._2 = l._2 + r._2;
  return ret;
}
```

Figure 4.3: `singleton`, which counts up the siblingless nodes in a given tree.

#### Suppressing Code Generation

Our parallelizer generates a C++ program file encapsulated for `#include` from a given C program. Basically, it contains all symbols in the given C program. This dirties the namespaces in which generated programs are included. For example, the type `BNint` would be used only for describing the function `mis`. After debugging `mis` as a sequential C program, we would not use `BNint` anymore. Then, `BNint` is redundant for the includers of its generated program. To clean up these redundant symbols, our parallelizer provides the API for suppressing code generation. Through the syntax of `#pragma exclude name1 name2` ..., we can exclude the symbols `name1`, `name2`, ... from generated programs.

As a tricky application of this functionality, we can overwrite symbols with macros in the side of includers. For example, we can redefine `max` and `NINF` as macros in the includer side of the program generated from the program described in Figure 4.5.

### 4.3.3 Algorithm Descriptions

We describe the expressiveness of and restrictions on algorithm descriptions that our parallelizer accepts.

#### Expressiveness of Algorithm Descriptions

Recursive functions like `mis` in Figure 4.1 are a typical description for algorithms on binary trees. In addition to such a typical description, our parallelizer accepts several kinds of algorithm descriptions.

For example, `singleton` described in Figure 4.3 calculates the number of the singleton (i.e., siblingless) nodes of a given binary tree. It adopts a style that tests each kind of nodes successively. Figure 4.4 shows another style of traversal on binary trees. `left_lean` traverses a given binary tree in a *left-leaning* manner: if an internal node has no left child, its right child is not traversed. This traversal assumes the case where every internal node has a left child.

Our parallelizer accepts the traversal on full binary trees as shown in Figure 4.5. `mis_fbt` returns unspecified value in the case where internal nodes have a single child. That is, `mis_fbt` is partially defined. Our parallelizer generates and uses operators that raise runtime exceptions for undefined cases. Entry functions generated thus handle full binary trees successfully and fail to handle general binary trees safely. We call operators for undefined cases *halt operators*.

```
#pragma pure all any
extern int all(int);
extern int any(int);

pair_int left_lean(BNint *t) {
  pair_int ret, l = {1,0}, r = {1,0};
  if (t->l != NULL)  {
    l = left_lean(t->l);
    if (t->r != NULL) {
      r = left_lean(t->r);
    }
  }
  ret._1 = all(t->v) & l._2 & r._2;
  ret._2 = any(t->v) | l._1 | r._1;
  return ret;
}
```

Figure 4.4: `left_lean`, which performs a left-leaning traversal. Pure functions `all` and `any` are used as predicates of payload values.

#### Restrictions on Algorithm Descriptions

Our parallelizer has three kinds of restrictions on algorithm descriptions.

The first is derived from parallel computing. For example, impure function calls and write to global variables are not allowed.

The second is derived from the formalization described of operator derivation. The `struct` types of return values have to consists of a constant number of fields of the same type. The condition part of if statements must not use the results of recursive calls. The descent and ascent in recursive calls must be moves between parents and children.

The third is derived from the implementation of our parallelizer. It includes miscellaneous restrictions for simplifying case extraction and symbolic execution described in Section 4.3.4. For example, the `struct` for nodes consists only of the payload value `v`, the left-child link `l`, and the right-child link `r`. Target recursive functions must not take `NULL`. Testing the kinds of nodes is limited to `NULL` checking of child links. Control structures are limited to if statements. Although these restrictions are rather severe, we consider that they are overall reasonable, assuming the first kind and second kind of restrictions.

### 4.3.4 Implementation Overview

We implemented our parallelizer on top of COINS[1]. Program analysis and transformation were implemented at the HIR level, where the AST of a given C program almost remains as the HIR. Figure 4.6 shows the compilation pipeline of our parallelizer. In the rest of this subsection, we describe each compilation step briefly along the pipeline described in Figure 4.6 by using the program described in Figure 4.1 as a running example.

#### Pragma Processing

After the C frontend construct HIRs, we first find and interpret `#pragma`.

If any `#pragma commSemiring` is not given, our parallelizer uses only built-in semirings. If a user-defined function is used in `#pragma commSemiring`, we generate a unique operator for the function and then replace its function calls with binary expressions internally in HIRs. As a result, we can handle the expressions over all commutative semirings uniformly on HIRs. Note that binary expressions with

[1] `http://coins-compiler.sourceforge.jp/`

```
#pragma commSemiring max "+" NINF 0
#pragma exclude max NINF
extern int NINF;
extern int max(int a, int b);

pair_int mis_fbt(BNint *t) {
  pair_int ret;
  if (t->l == NULL && t->r == NULL) {
    ret._1 = NINF;
    ret._2 = 0;
  } else {
    pair_int l = mis_fbt(t->l),
             r = mis_fbt(t->r);
    ret._1 = t->v + l._2 + r._2;
    ret._2 = max(l._1,l._2) + max(r._1,r._2);
  }
  return ret;
}
```

Figure 4.5: `mis_fbt`, which calculates the maximum independent sum of a given full binary tree.

generated operators are transformed back to function calls just before code generation. For example, after our parallelizer processes `#pragma commSemiring` in Figure 4.1, the `max` function is interpreted as an operator. Then, our parallelizer assumes that the `max` function and the `+` operator consist of a commutative semiring.

The symbols declared `#pragma pure` and `#pragma exclude` are registered in this step. The former information is used in matrix extraction and the latter information is used in code generation.

### Case Extraction

Our parallelizer tries to parallelize all given function definitions. It first constructs from each target definition, program slices in four cases that correspond to the four primary operators for reduce over general binary trees (see Section 4.4). Specifically, the slice $s_{00}$ in the case of no child (i.e., *Leaf*), the slice $s_{10}$ in the case of only the left child (i.e., *Left*), the slice $s_{01}$ in the case of only the right child (i.e., *Right*), and the slice $s_{11}$ in the case of both children (i.e., *Branch*) are constructed.

In the rest of compilation process, we focus on the `mis` function definition. For example, the following is the slice $s_{10}$ constructed from `mis`.

```
pair_int mis(BNint *t) {
  pair_int ret, l = {NINF,0}, r = {NINF,0};
  l = mis(t->l);
  ret._1 = t->v + l._2 + r._2;
  ret._2 = max(l._1, l._2) + max(r._1, r._2);
  return ret;
}
```

In this step, we require consideration of short-circuit logical operators `&&` and `||` in COINS. The C frontend translates both into if statements containing goto/label statements. Because the goto/label statements complicate control flow, we eliminate them by replicating the then/else part of if statements containing them. As a result, the implementation of case extraction becomes simple.

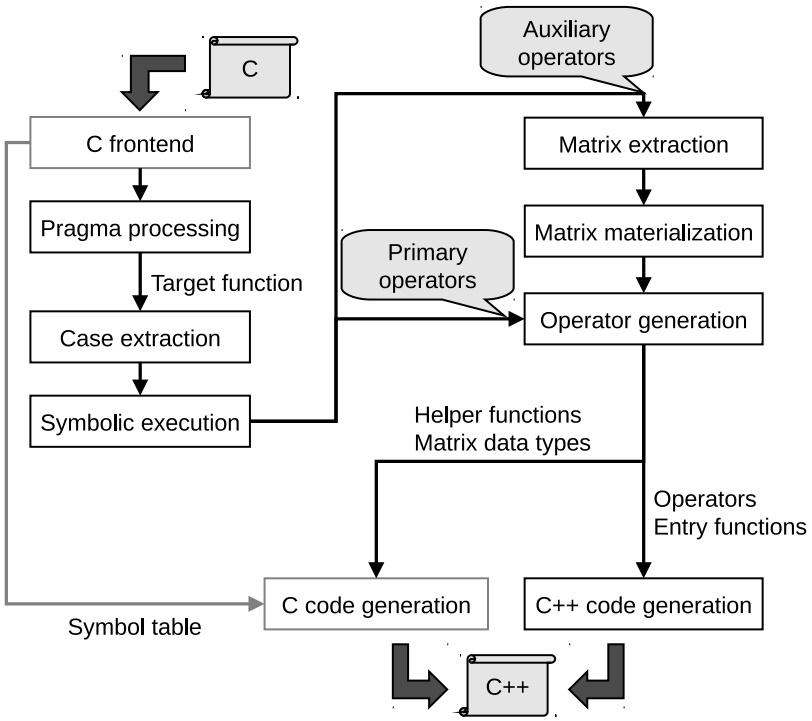



Figure 4.6: Compilation pipeline of our parallelizer.

### Symbolic Execution

Program slices represent computations in the form of statements. Computations in the form of pure expressions are appropriate to operators for skeletons. We therefore transform the program slices $s_{00}$, $s_{01}$, $s_{10}$, and $s_{11}$ respectively into expressions $e_{00}$, $e_{01}$, $e_{10}$, and $e_{11}$ through symbolic execution.

Specifically, symbolic execution is to update a symbolic environment (i.e., a mapping from variables to expressions) in a step-by-step abstract interpretation where assignments cause symbolic substitution and if statements construct ternary conditional expressions. The resultant expression is a return expression to which we apply symbolic substitution based on a final symbolic environment.

After we obtain a single expression corresponding to a slice, we abstract the payload value of a current node, the result of the recursive call for a left child, and that for a right child, by using special variables `v`, `l`, and `r`. These special variables correspond to the formal parameters of operators for skeletons. Finally, constructed expressions $e_{00}$, $e_{01}$, $e_{10}$, and $e_{11}$ become equivalent to the body expressions of the primary operators $k_{00}$, $k_{01}$, $k_{10}$, and $k_{11}$ for reduce over general binary trees (see Section 4.4).

For example, the $s_{10}$ of `mis` is transformed into the following $e_{10}$:

```
_pair_int(v + l._2 + 0, max(l._1, l._2) + max(NINF, 0)),
```

where the function `_pair_int` is a helper function that works as the constructor of `pair_int`. Our parallelizer generates such helper functions for convenience.

After $e_{00}$, $e_{01}$, $e_{10}$, and $e_{11}$ are obtained, we validate them. If we find local variables in expressions, these values are unspecified. We regard such expressions as undefined cases and do not use the expressions themselves in later steps. We, however, register undefined cases for generating halt operators.

### Matrix Extraction

To obtain auxiliary operators, we have to formalize a target function as a multilinear computation. This corresponds to transforming the primary operators used for internal nodes, i.e., $e_{01}$, $e_{10}$, and $e_{11}$ into the

matrix-vector product like the equation (4.1). That is, we have to extract coefficient matrices.

By assuming the linearity of expressions, we can implement the extraction of coefficient matrices simply as partial differentiation over a commutative semiring. If we obtain nonlinear a partial derivative, we judge that matrix extraction fails. That is, partial differentiation also works as linearity checking.

Specifically, we extract four coefficient matrices $A_l$, $A_r$, $A_{\bar{l}}$, and $A_{\bar{r}}$ that are respectively used in the auxiliary operators $\psi_l$, $\psi_r$, $\psi_{\bar{l}}$, and $\psi_{\bar{r}}$ for reduce over general binary trees (see Section 4.4). $A_l$ consists of the partial derivatives of $e_{11}$ with respect to `l`; $A_r$ consists of the partial derivatives of $e_{11}$ with respect to `r`; $A_{\bar{r}}$ consists of partial derivative of $e_{01}$ with respect to `r`; $A_{\bar{l}}$ consists of the partial derivatives of $e_{10}$ with respect to `l`.

More precisely, augmented coefficient matrices in the equation (4.2) are extracted. The extraction of the part of $\mathbf{b}_i$ is immediate. It is sufficient to substitute `l` or `r` with a null vector. For example, from the $e_{10}$ of `mis`, we can extract the following augmented matrix:

$$\begin{pmatrix} \texttt{NINF} & \texttt{v} & \texttt{NINF} \\ 0 & 0 & \texttt{NINF} \\ \texttt{NINF} & \texttt{NINF} & 0 \end{pmatrix}.$$

The $(1, 2)$-component here is the partial derivative of the first member `v + l._2 + 0` of $e_{10}$, with respective to the second member `l._2` of `l`.

Our parallelizer tries matrix extraction successively with respect to each commutative semiring until all coefficient matrices are successfully obtained. In this step, our parallelizer also performs constant folding over a target commutative semiring. Although this constant folding would improve the runtime performance of generated operators, its main aim is to simplify the construction of abstract matrices in matrix materialization.

### Matrix Materialization

Extracted matrices often consist much of 0 and 1 over a semiring. In particular, augmented coefficient matrices, which our parallelizer actually extracts, contain 0 and 1 by definition. Some components of such matrices remain 0 or 1 in matrix-matrix multiplication. Then, we can eliminate such invariant components from dynamic (i.e., runtime) data and instead embed constants into the code of auxiliary operators. In the step of matrix materialization, we find such invariant components and define the concrete data types of matrices.

To determine invariant components of 0 or 1, we have only to distinguish 0, 1, and dynamic values, i.e, the values of variables. Letting $V$ be the abstract value of variables, we consider a three-valued commutative semiring $(\{0, 1, V\}, +, \times, 0, 1)$, where $V + a = V$ for any $a \in \{0, 1, V\}$ and $V \times V = V$. We construct abstract matrices over $(\{0, 1, V\}, +, \times, 0, 1)$ from extracted matrices. Then, the supremum of all possible matrix products derived from the abstract matrices suffices for determining necessary components as dynamic data. To calculate supremums, we define the join operator $\sqcup$ over $\{0, 1, V\}$ as $V \sqcup c = V$ and $c \sqcup c' = V$ for any $c \in \{0, 1\}$. In summary, we can calculate the supremum of abstract matrix products as the least fixed point of matrix multiplication over $(\{0, 1, V\}, +, \times, 0, 1)$ and matrix updating over $(\{0, 1, V\}, \sqcup)$ with all abstract matrices.

For example, the supremum of abstract matrix products derived from augmented coefficient matrices of `mis` is the following:

$$\begin{pmatrix} V & V & 0 \\ V & V & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Then, the four components of $V$ suffices for the dynamic data of a matrix. Our parallelizer therefore defines the data type of matrices as the following `struct`.

```
struct _mis_mat {
 int _0_0, _0_1, _1_0, _1_1;
};
```

The member names here correspond to the zero-based numbering of component indices.

**Operator Generation and Code Generation**

Since $e_{00}$, $e_{01}$, $e_{10}$, and $e_{11}$ are equivalent to their bodies, it is immediate to generate the primary operators. After augmented coefficient matrices and matrix data types are obtained, we can generate auxiliary operators simply by using symbolic matrix multiplication, following the general forms described in Section 4.2. Note that the special variables `v`, `l`, and `r` are captured as the formal parameters of generated operators.

The final code generation is straightforward. Because of high compatibility between C89 and C++03, we have been able to reuse the C code generator of COINS for code generation. The code generation of helper functions and matrix data types is completely performed by the C code generator. We implemented an ad hoc C++ code generator for entry functions and operator definitions as a simple extension of the C code generator because they necessitate C++ functionality.

From the augmented coefficient matrix above of `mis`, generated is the following function (more precisely, a member function of C++ function objects) that corresponds to $\psi_{\bar{r}}$.

```
hir_t_void operator()(struct _mis_mat &m, const hir_t_int v) const
{
  hir_s_private hir_v_auto hir_t_int _var43;
  hir_s_private hir_v_auto hir_t_int _var45;
  hir_s_private hir_v_auto hir_t_int _var47;
  hir_s_private hir_v_auto hir_t_int _var49;
  _var43 = (hir___ADD(v,m._1_0));
  _var45 = (hir___ADD(v,m._1_1));
  _var47 = max( m._0_0,m._1_0);
  _var49 = max( m._0_1,m._1_1);
  m._0_0 = _var43;
  m._0_1 = _var45;
  m._1_0 = _var47;
  m._1_1 = _var49;
}
```

The matrix argument `m` is overwritten with a return value because passed data is unnecessary in reduce.

Right before code generation, our parallelizer excludes symbols inconvenient to C++ programs. For example, `wchar_t` is a `typedef`ed type in C but a keyword in C++. If `wchar_t` were generated as C, resultant programs would have a syntactic error in C++. We therefore have to exclude `wchar_t` from code generation. Similarly, we exclude symbols specified in `#pragma exclude` from code generation.

## 4.4 Underlying Binary Tree Skeleton

As mentioned in Section 4.3, recursive functions in a conventional style of C lead to recursive computations over general binary trees. We therefore use reduce over general binary trees for the underlying skeleton of entry functions generated. In this section, we describe the definition, the specialization, and the implementation of reduce over general binary trees.

### 4.4.1 Definition

We first define general (i.e., non-full) binary trees as the following grammar:

$$
\begin{aligned}
GBinTree_\alpha &= Branch(x, GBinTree_\alpha, GBinTree_\alpha),\\
GBinTree_\alpha &= Left(x, GBinTree_\alpha),\\
GBinTree_\alpha &= Right(x, GBinTree_\alpha),\\
GBinTree_\alpha &= Leaf(x).
\end{aligned}
$$

*Left* denotes a node having only the left child and *Right* denotes a node having only the right child. We can define reduce over $GBinTree_\alpha$ as

$$\begin{aligned}
\mathsf{reduce}(k_{11}, k_{10}, k_{01}, k_{00}, \mathit{Branch}(x, l, r)) &= k_{11}(\mathsf{reduce}(k_{11}, k_{10}, k_{01}, k_{00}, l), x, \mathsf{reduce}(k_{11}, k_{10}, k_{01}, k_{00}, r)),\\
\mathsf{reduce}(k_{11}, k_{10}, k_{01}, k_{00}, \mathit{Left}(x, l)) &= k_{10}(\mathsf{reduce}(k_{11}, k_{10}, k_{01}, k_{00}, l), x),\\
\mathsf{reduce}(k_{11}, k_{10}, k_{01}, k_{00}, \mathit{Right}(x, r)) &= k_{01}(x, \mathsf{reduce}(k_{11}, k_{10}, k_{01}, k_{00}, r)),\\
\mathsf{reduce}(k_{11}, k_{10}, k_{01}, k_{00}, \mathit{Leaf}(x)) &= k_{00}(x).
\end{aligned}$$

Parameters $k_{11}$, $k_{10}$, $k_{01}$, and $k_{00}$ are respectively the interpretations of *Branch*, *Left*, *Right*, and *Leaf*.

We can define the segmented version of $GBinTree_\alpha$ as

$$\begin{aligned}
GSegTree_\alpha &= \mathit{Branch}(GContext_\alpha, GSegTree_\alpha, GSegTree_\alpha),\\
GSegTree_\alpha &= \mathit{Left}(GContext_\alpha, GBinTree_\alpha),\\
GSegTree_\alpha &= \mathit{Right}(GContext_\alpha, GBinTree_\alpha),\\
GSegTree_\alpha &= \mathit{Leaf}(GBinTree_\alpha),\\
GContext_\alpha &= \mathit{Branch}_1(x, GContext_\alpha, GBinTree_\alpha),\\
GContext_\alpha &= \mathit{Branch}_2(x, GBinTree_\alpha, GContext_\alpha),\\
GContext_\alpha &= \mathit{Left}(x, GContext_\alpha),\\
GContext_\alpha &= \mathit{Right}(x, GContext_\alpha),\\
GContext_\alpha &= \mathit{Hole}(x).
\end{aligned}$$

We can also define reduce over $GSegTree_\alpha$.

$$\begin{aligned}
&\mathsf{reduce} : (b \times a \times b \to b) \times (b \times a \to b) \times (a \times b \to b) \times (a \to b)\\
&\qquad \times (a \to c) \times (b \times c \times b \to b) \times (b \times c \to b) \times (c \times b \to b)\\
&\qquad \times (c \times c \times b \to c) \times (b \times c \times c \to c) \times (c \times c \to c) \times (c \times c \to c)\\
&\qquad \times GBinTree_\alpha \to b\\
&\mathsf{reduce}(k_{11}, k_{10}, k_{01}, k_{00}, \tau, \phi_{11}, \phi_{10}, \phi_{01}, \psi_l, \psi_r, \psi_{\bar{l}}, \psi_{\bar{r}}, t) = \mathit{red}(t),\\
&\quad \text{where } \mathit{red}(\mathit{Branch}(x, l, r)) = \phi_{11}(\mathit{red}(l), \mathit{redCtx}(c), \mathit{red}(r))\\
&\qquad \mathit{red}(\mathit{Left}(c, l)) = \phi_{10}(\mathit{red}(l), \mathit{redCtx}(c))\\
&\qquad \mathit{red}(\mathit{Right}(c, r)) = \phi_{01}(\mathit{redCtx}(c), \mathit{red}(r))\\
&\qquad \mathit{red}(\mathit{Leaf}(t)) = \mathsf{reduce}(k_{11}, k_{10}, k_{01}, k_{00}, t)\\
&\qquad \mathit{redCtx}(\mathit{Branch}_1(x, l, r)) = \psi_l(\mathit{redCtx}(l), \tau(x), \mathsf{reduce}(k_{11}, k_{10}, k_{01}, k_{00}, r)),\\
&\qquad \mathit{redCtx}(\mathit{Branch}_2(x, l, r)) = \psi_r(\mathsf{reduce}(k_{11}, k_{10}, k_{01}, k_{00}, l), \tau(x), \mathit{redCtx}(r)),\\
&\qquad \mathit{redCtx}(\mathit{Left}(x, l)) = \psi_{\bar{l}}(\mathit{redCtx}(l), \tau(x)),\\
&\qquad \mathit{redCtx}(\mathit{Right}(x, r)) = \psi_{\bar{r}}(\tau(x), \mathit{redCtx}(r))\\
&\qquad \mathit{redCtx}(\mathit{Hole}(x)) = \tau(x),
\end{aligned}$$

and algebraic conditions are as follows:

$$
\begin{aligned}
k_{11}(x,l,r) &= \phi_{11}(l,\tau(x),r),\\
\phi_{11}(\phi_{11}(l',y',r'),y,r) &= \phi_{11}(l',\psi_l(y',y,r),r'),\\
\phi_{11}(\phi_{10}(l',y'),y,r) &= \phi_{10}(l',\psi_l(y',y,r)),\\
\phi_{11}(\phi_{01}(y',r'),y,r) &= \phi_{01}(\psi_l(y',y,r),r'),\\
\phi_{11}(l,y,\phi_{11}(l',y',r')) &= \phi_{11}(l',\psi_r(l,y,y'),r'),\\
\phi_{11}(l,y,\phi_{10}(l',y')) &= \phi_{10}(l',\psi_r(l,y,y')),\\
\phi_{11}(l,y,\phi_{01}(y',r')) &= \phi_{01}(\psi_r(l,y,y'),r'),\\
k_{10}(l,x) &= \phi_{10}(l,\tau(x)),\\
\phi_{10}(\phi_{11}(l',y',l'),y) &= \phi_{11}(l',\psi_{\bar{l}}(y',y),r'),\\
\phi_{10}(\phi_{10}(l',y'),y) &= \phi_{10}(l',\psi_{\bar{l}}(y',y)),\\
\phi_{10}(\phi_{01}(y',r'),y) &= \phi_{01}(\psi_{\bar{l}}(y',y),r'),\\
k_{01}(x,r) &= \phi_{01}(\tau(x),r),\\
\phi_{01}(y,\phi_{11}(l',y',r')) &= \phi_{11}(l',\psi_{\bar{r}}(y,y'),r'),\\
\phi_{01}(y,\phi_{10}(l',y')) &= \phi_{10}(l',\psi_{\bar{r}}(y,y')),\\
\phi_{01}(y,\phi_{01}(y',r')) &= \phi_{01}(\psi_{\bar{r}}(y,y'),r').
\end{aligned}
$$

Auxiliary operators $\phi_{11}$, $\phi_{10}$, $\phi_{01}$, $\psi_l$, $\psi_r$, $\psi_{\bar{l}}$, and $\psi_{\bar{r}}$ are respectively the interpretations of *Branch*, *Left*, and *Right* for $GSegTree_\alpha$, and $Branch_1$, $Branch_2$, *Left*, and *Right* for $GContext_\alpha$.

$\psi_{\bar{l}}$ and $\psi_{\bar{r}}$ correspond to the COMPRESS operation from the perspective of tree reduction. However, $\psi_{\bar{l}}$ and $\psi_{\bar{r}}$ contain matrix construction, which corresponds to the RAKE operation. We can therefore consider that $\psi_{\bar{l}}$ and $\psi_{\bar{r}}$ respectively correspond to $\text{SHUNT}_l$ and $\text{SHUNT}_r$, where the RAKE operation is applied to a missing leaf in the form of matrix construction with no value of recursive calls.

To best our knowledge, the skeleton identical to reduce over $GBinTree_\alpha$ has not been presented. However, we can derive its definition (including its algebraic conditions) straightforwardly from the notion of segmented trees and that of tree contraction operations.

### 4.4.2 Specialization to Multilinear Computations

We can specialize reduce over $GBinTree_\alpha$ to multilinear computations. Let $\mathcal{V}_\beta$ be the type of vectors whose components are of type $\beta$ and $\mathcal{M}_\beta$ be the type of matrices whose components are of type $\beta$.

First, from the definition (4.1) of multilinear functions, we can determine the types of the primary operators as follows.

$$
\begin{aligned}
&k_{11} : \mathcal{V}_\beta \times \alpha \times \mathcal{V}_\beta \to \mathcal{V}_\beta\\
&k_{10} : \mathcal{V}_\beta \times \alpha \to \mathcal{V}_\beta\\
&k_{01} : \alpha \times \mathcal{V}_\beta \to \mathcal{V}_\beta\\
&k_{00} : \alpha \to \mathcal{V}_\beta
\end{aligned}
$$

Next, as mentioned in Section 4.2, we consider extended payload values of type $\alpha \times \mathcal{M}_\beta$. On the basis of the meaning of extended payload values, we can define the following auxiliary operators:

$$
\begin{aligned}
&\phi_{11} : \mathcal{V}_\beta \times (\alpha \times \mathcal{M}_\beta) \times \mathcal{V}_\beta \to \mathcal{V}_\beta\\
&\phi_{11}(\mathbf{y}_l,(x,M),\mathbf{y}_r) = Mk_{11}(\mathbf{y}_l,x,\mathbf{y}_r),\\
&\phi_{10} : \mathcal{V}_\beta \times (\alpha \times \mathcal{M}_\beta) \to \mathcal{V}_\beta\\
&\phi_{10}(\mathbf{y}_l,(x,M)) = Mk_{10}(\mathbf{y}_l,x),\\
&\phi_{01} : (\alpha \times \mathcal{M}_\beta) \times \mathcal{V}_\beta \to \mathcal{V}_\beta\\
&\phi_{01}((x,M),\mathbf{y}_r) = Mk_{01}(x,\mathbf{y}_r).
\end{aligned}
$$

The operators above are simple instances of operators for reduce and not specialized instances because we obtain them through type substitution.

We can specialize the definitions of $\psi_l$, $\psi_r$, $\psi_{\bar{l}}$, and $\psi_{\bar{r}}$. An important point of specialization is that we introduce extended payload values only for remaining the payload values of hole nodes. If we store on its root the payload value of the hole of a one-hole context, we do not have to introduce extended payload values. In this case, the four auxiliary operators can be simplified as follows:

$$\begin{aligned}
&\psi_l : \mathcal{M}_\beta \times \alpha \times \mathcal{V}_\beta \to \mathcal{M}_\beta \\
&\psi_l(M, x, \mathbf{y}_r) = A_l M, \\
&\psi_r : \mathcal{V}_\beta \times \alpha \times \mathcal{M}_\beta \to \mathcal{M}_\beta \\
&\psi_r(\mathbf{y}_l, x, M) = A_r M, \\
&\psi_{\bar{l}} : \mathcal{M}_\beta \times \alpha \to \mathcal{M}_\beta \\
&\psi_{\bar{l}}(M, x) = A_{\bar{l}} M, \\
&\psi_{\bar{r}} : \alpha \times \mathcal{M}_\beta \to \mathcal{M}_\beta \\
&\psi_{\bar{r}}(x, M) = A_{\bar{r}} M,
\end{aligned}$$

where $x$, $\mathbf{y}_l$, and $\mathbf{y}_r$ are implicitly used in $A_l$, $A_r$, $A_{\bar{l}}$, and $A_{\bar{r}}$.

Then, the auxiliary operator $\tau$ to lift payload values becomes unnecessary because we reduce a one-hole context in a sequential bottom-up manner. Instead, we lift every hole node to the identity matrix $I$. We thus can reduce an one-hole context to a single matrix by using $\psi_l$, $\psi_r$, $\psi_{\bar{l}}$, $\psi_{\bar{r}}$, and the primary operators. We construct an extended payload value for each hole node by pairing the matrix to which a one-hole context reduces and the payload value of its hole. After that, we reduce a global tree by using $\phi_{11}$, $\phi_{10}$, and $\phi_{01}$ successfully.

### 4.4.3 Library Implementation

We implemented reduce over $GSegTree_\alpha$ as a C++ library with MPI[2], on the basis of an array-based implementation [Mat07a] of tree skeletons. Our implementation is a straightforward extension of the original implementation [Mat07a] for full binary trees to that for general binary trees.

Our library implementation utilizes C++ templates extensively. C++ templates enabled us to implement the specialization to multilinear computations easily. What we did for the specialization was only to specialize the reduction of one-hole contexts. After including it in the basic implementation, the instantiation with specialized operator types guides reduce to the specialized implementation.

Our library implementation assumes programming based upon the SPMD (single-program multiple-data) model with MPI, as in existing data-parallel skeleton libraries [SM14a, EM14]. The functions `config::init` and `config::finzalize` that our library provides (see Figure 4.2) are indeed the wrappers of `MPI_init` and `MPI_finalize`.

Our library is assumed to be used only from entry functions. Entry functions are desired to select an appropriate implementation of skeletons from user's input. Because entry functions, where generated operators are used implicitly from programmers, take only a binary tree data structure, it is natural that the implementations of binary tree data structures specify those of skeletons. If we can implement entry functions as generic functions, the code generation of our parallelizer becomes simple and the generated programs become convenient in C++. If there is a clear interface between entry functions and tree skeletons, we can achieve loose coupling between our parallelizer and our library. To achieve these, we have designed our library to enable calling reduce via tree data structures. As a result, we can implement an entry function `f` as follows.

```
template <class BinTree>
struct vector_type f(const BinTree& t)
{
  return BinTree::skeletons::reduce(/* operators omitted */, t)
}
```

[2] `http://www.mpi-forum.org/`

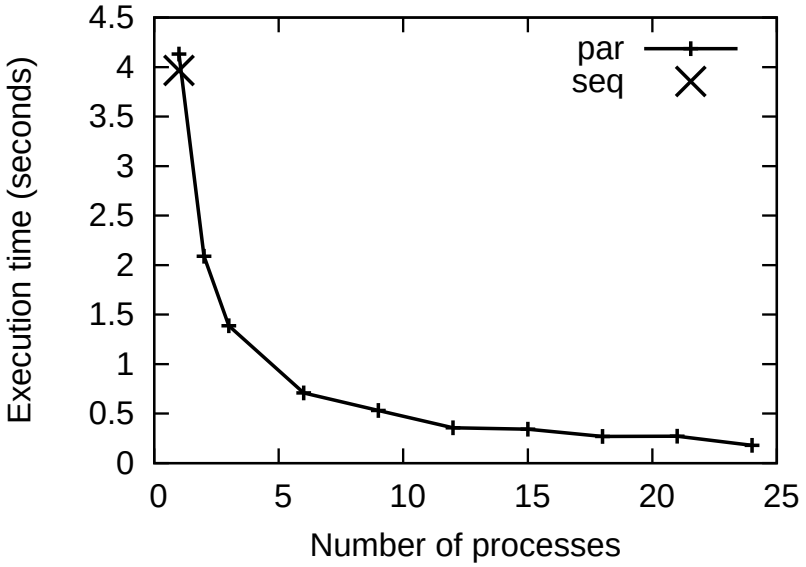



Figure 4.7: Execution time of reduce for `mis` described in Figure 4.1, where `seq` means reduce over $GBinTree_{\texttt{int}}$ and `par` means reduce over $GSegTree_{\texttt{int}}$. Time for tree construction is not contained.

On the basis of the genericity of templates, $f$ is generic with respect to the parameter type *BinTree*. Given *BinTree* determines the implementation of reduce.

## 4.5 Preliminary Experiments on Parallel Execution

We tested the parallel execution based on our underlying skeleton library. We measured the execution time of the entry function generated from `mis` described in Figure 4.1.

We used a three-node cluster each of whose nodes was equipped with two AMD Opteron 2376 (2.3 GHz, 4 cores) processors and 8-GB memory (ECC DDR2-667) and connected to the others with 1000BASE-T. Each node was running Linux 3.2.0 (64-bit). We used g++ 4.6.3 with the O3 optimization for native compilation and OpenMPI 1.4.3.

As an input, we used a randomly generated tree that had $2^{28}$ nodes. To make it easy to expect execution time, we made the segment sizes of its segmented version almost the same. The number of the segments was 48. Figure 4.7 shows the measurements of the execution time of reduce over $GBinTree_{\texttt{int}}$ (`seq` for short) and that of reduce over $GSegTree_{\texttt{int}}$ (`par` for short).

The results conformed with our expectation. `par` with a single process was a little slower than `seq`. We attribute this slowdown to auxiliary operators $\psi_l$, $\psi_r$, $\psi_{\bar{l}}$, and $\psi_{\bar{r}}$. These perform matrix-matrix multiplication, while primary operators $k_{11}$, $k_{10}$, and $k_{01}$ perform matrix-vector multiplication. This means that `par` has unavoidable overhead. This overhead is proportional to the height of an input tree and the size of matrices. Although the overhead that we observed experimentally was small and acceptable, this quantity is not guaranteed in general.

## 4.6 Benefits of Hiding Tree Skeletons

The main aim of our parallelizer is to hide tree skeletons from programmers. In this section, we describe the benefits and importance of hiding tree skeletons.

Table 4.1: Relationship between degree of trees and entanglement of operators for reduce.

| | Full binary | Full $k$-ary | Binary | $k$-ary |
|---|---|---|---|---|
| No. of primary operators | 2 | 2 | 4 | $2^k$ |
| No. of auxiliary operators | 4 | k+2 | 8 | $(k+2)2^{k-1}$ |
| No. of algebraic conditions | 3 | k+1 | 15 | $(k2^{k-1}+1)(2^k-1)$ |

### 4.6.1 Entanglement of Operators for Tree Skeletons

The main reason to hide tree skeletons is that operators for tree skeletons are too complicated for programmers. The algebraic conditions of reduce, uAcc, and dAcc over $SegTree_\alpha$ are actually difficult as mentioned in Section 4.1. However, the difficulty is more than that. The number of both operators and algebraic conditions on them, which we call the *entanglement* of operators, sharply increases in proportion to the degree of trees. This is a serious problem.

Table 4.1 shows the entanglement of the operators for reduce straightforwardly generalized to $k$-ary trees. As seen from Table 4.1 and the definition of reduce over $GSegTree_\alpha$, the entanglement of operators in the case of general $k$-trees is terrible and unrealistic for human use.

As seen from the definitions of reduce, each of most operators correspond to each kind of nodes. To be as general definitions as possible, higher-order functions have to take distinct operators for all kinds of nodes. Since any full $k$-ary tree has only two kinds of nodes, the case of full $k$-ary trees is relatively tractable. This situation is not limited to tree skeletons. However, the transformation of a tree into its segmented version multiplies the number of the kinds of nodes. Moreover, the number of algebraic conditions corresponds roughly to the number of the possible parent-child pairs of kinds. The number of the kinds of the nodes of a give tree therefore has a big impact on the entanglement of operators.

If we describe algorithms on trees as recursive functions in a conventional style like Figure 4.1, we do not confront this problem. Although the code of a recursive function becomes long in proportion to the degree of given trees, the same calculation for different kinds of nodes can share its code. For example, `mis` in Figure 4.1 shares the calculation for the four kinds of nodes, except for defining `l` and `r`. The expressiveness of recursive functions thus tames the complication of APIs.

### 4.6.2 Trade-off between Generality and Simplicity

As seen from Table 4.1, it is adequate for human use that existing tree skeletons were based on full binary trees [Ski96, GCS94, Mat07b]. If we would like to use more complicated trees than full binary trees, these are, however, troublesome. In such cases, we have to encode a desired tree into a full binary tree and implement a desired computation by using skeletons over full binary trees. This is a problem both on efficiency and usability.

The implementers of tree skeletons cannot determine trees that the users desire because trees are versatile and flexible. The implementers would design tree skeletons as generally for versatile use as possible. As a result, the API of tree skeletons becomes too general and complicated. Even though the implementers could provide a simple and useful API for a specific tree, such ad hoc extensions do not lead to a well-designed skeleton library. In designing APIs, there is generally a tradeoff between generality and simplicity. It is therefore difficult to obtain both the generality for versatile use and the simplicity for easy use.

### 4.6.3 Abstraction Layer between Specifications and Skeletons

The cause of this tradeoff is that the API of skeletons is the boundary between the abstraction layer of specifications and that of (the implementation of) skeletons. A conflict of interest between the users in the layer of specifications and the implementers in the layer of skeletons causes this tradeoff. Our parallelizer works as an abstraction layer that defuses this conflict.

The abstraction layer by our parallelizer promotes loose coupling between specifications and skeletons. A high degree of freedom in specifications enables programmers to describe algorithms as recursive functions in a conventional style of C, which do not impose complicated APIs. A high degree of freedom

in skeletons enables the adoption of general but complicated skeletons and moreover their specialization to multilinear computations. In this sense, the abstraction layer by our parallelizer has successfully granted both the request of the users and that of the implementers.

### 4.6.4 Implicitly Skeletal Programming on Trees

Skeletons are usually designed as abstractions for human use. Tree skeletons, however, are not appropriate to human use because of the generality of computations on trees. They are, in fact, appropriate to high-level background services for parallel computing that are used in the programs generated from user-friendly specifications. Such an indirect use of tree skeletons is promising.

In contrast to a usual *explicit* approach where programmers use skeletons directly, we call such indirect use of skeletons an *implicit* approach. In an implicit approach, we can also enjoy the benefits of high-level abstraction derived from skeletons. The compilers of specification languages do no have to take account of the low-level details of parallel machines and generated programs with skeletons have high portability with performance guarantee.

In summary, our true aim of this work is to present this implicit approach in programming with tree skeletons (i.e., implicitly skeletal programming on trees) and our parallelizer is a successful implementation of the presented approach.

## 4.7 Related Work

We use the formalization by Matsuzaki et al. [MHT06] of systematic derivation of auxiliary operators from primary operators. Although we slightly generalize their formalization for general $k$-ary trees, this generalization is immediate. They also developed a prototype implementation of an operator generator based on their formalization. Although it was able to generate operators for reduce automatically from a recursive function described in an imperative language, its aim was to demonstrate the feasibility of their formalization. It was not designed to work as an abstraction for parallel programming.

Automatic derivation of the operators for list skeletons was studied [MHT08, MMM$^+$07, SI11a, XKH04]. Our work is most relevant to the work [XKH04] by Xu et al. Their system guaranteed the linearity of given recursive functions on lists by using type inference on the basis of given semirings, and then generated a list homomorphism [Bir87], which is a kind of list skeletons, from typed terms. Their type inference and type-directed program transformation is equivalent to our matrix extraction.

The automatic parallelization based on quantifier elimination [MM10] can deal theoretically with recursions on trees. Their prototype implementation, however, dealt only with recursions on lists and therefore the feasibility of their method for automatic parallelization of recursions on trees is still unknown. The procedure of quantifier elimination is much more expensive than partial differentiation. Their method is therefore heavyweight for compiler integration, while our parallelizer has been successfully integrated into a C compiler.

The implicit use of skeletons is not rare and can be found in existing work. MapReduce [DG04] is the most popular (or de facto standard) data-parallel skeleton for data processing on large-scale clusters. The MapReduce system was implemented in C++ for utilizing user-defined C++ functions. Although the main benefit of MapReduce originally claimed was simplicity, its usability left much room for improvement. As a result, Sawzall [PDGQ05], which was a typed scripting language for querying, FlumeJava [CRP$^+$10], which was a parallel collection library in Java, and Dremel [MGL$^+$10], which was a query language like SQL were developed for hiding use of MapReduce. The generate-test-and-aggregate (GTA) framework [EFH12], which is a kind of skeletons over multisets (i.e., bags), was developed for the same purpose. Its library implementation [LEH14] used the MapReduce API of Spark [ZCD$^+$12] and Hadoop[3] on the inside. These studies used MapReduce as an infrastructure of data processing.

NESL [BCH$^+$94] is the most remarkable one of parallel languages that exploit parallelism on trees[4]. NESL is a typed functional language with no side effect. In NESL, the parallelism of list comprehension and recursive functions is implicitly exploited and programmers describe algorithms in a sequential

[3] `http://hadoop.apache.org/`

[4] It is called nested data parallelism in the context of NESL.

manner. In NESL, trees are operated as nested lists with recursive calls. By using flattening [Ble90], the parallelism in trees is transformed into the parallelism in arrays and finally tree computations is implemented with vector operations. This flattening incurs the time and space overhead of array representations themselves and operations on them. To tame this overhead, several studies dealt with the avoidance of flattening [KCL+12, BFR+13]. However, this overhead is the result of a weak abstraction of parallel computations on trees. The construction of and the computations on trees in NESL are respectively implemented as list constructions and list computations repeated in recursive functions. This means that NESL have to provide a list view of trees and enable arbitrary recursions. Therefore, the encapsulation of tree data structures is difficult and there is not much room for an efficient implementation. In contrast, tree skeletons can use an encapsulated implementation of trees and therefore have much room for an efficient implementation. Entry functions in our approach actually utilize this property.

## 4.8 Conclusion

We have presented our parallelizer for implicitly skeletal programming on trees. Our parallelizer builds on the existing formalization [MHT06] and enables programmers to enjoy the benefits of tree skeletons with no burden derived from their complication,

The current implementation deals only with reduce over binary trees. Further development is left for future work. An extension to the computations of uAcc and dAcc is straightforward. However, automatic parallelization of recursive functions that consist of a mixture of reduce, uAcc, and dAcc will be nontrivial. On the basis of the diffusion theorem [HTI99], we can parallelize such computations in theory. Its automation with reasonable program analysis to recursive functions is, however, unexplored. The computations over trees of unbounded degree are also an issue. The tree contraction algorithm and operations [MM11a] to trees of unbounded degree will be helpful.

The current implementation uses the only implementation of a tree skeleton. An implementation specialized to a specific composition of tree skeletons is, however, more efficient. In such cases, dispatch to an appropriate implementation of a composition of tree skeletons is desired. It is promising that underlying skeleton libraries in C++ implement this dispatch because C++ template metaprogramming enables such optimizations [ME10, EM14] at the library level. This approach will promote the separation of concerns between parallelizers and underlying skeleton libraries.

Recent work [Mor11, Mor12] by Morihata applied tree contraction algorithms successfully to attributed tree transducers and macro tree transducers, which are restricted recursive functions that transform given trees. These results theoretically demonstrate the possibility of parallelizing recursive functions to tree skeletons for tree transformations. Their practical usability is, however, still unknown and automatic parallelization based on these results is left for future work.

# Chapter 5

# Limitations of Tree Skeletons

In Chapters 3 and 4, we have improved the usability of tree skeletons both for skeleton implementers and skeleton users. However, tree skeletons are still not practically useful. Applications of tree skeletons are actually very limited. For example, the maximum marking problems [MHT08] on trees can be solved by using tree skeletons. Although maximum marking problems themselves can cover a class of valuable optimization problems, their settings to which we can apply tree skeletons easily are rather artificial. This is because we adopt an unreasonable approach to tree-based computations.

Trees in programming are more than trees. Trees in programming are abstract representations of some data. The interpretation of trees is closely relevant to this underlying data. In the case of join lists, the underlying data is lists to which the concatenation operation is constant-time. Typically, binary search trees are closely relevant to the underlying data. Their tree structures represent sorted sets. We can balance binary search trees by definition. When querying a binary search tree, we indeed traverse it but essentially calculate the magnitude relation between a key and a subset that its subtree represents.

The interpretation of $BinTree_\alpha$ is nothing more than an algebra over full binary trees. The interpretation of $SegTree_\alpha$ incorporates tree contraction operations but is also nothing more than an algebra over segmented trees. To use tree skeletons, we have to define these interpretations as algebra independent of the underlying data of trees. Intuitively, this is to implement, for example, the operations on binary search trees only with recursion on their structures without any comparison of their payload values. Such tree manipulations are obviously counter-intuitive and indubitably inefficient. Therefore, use of tree skeletons is unreasonable in programming for tree-based computations.

The consideration of underlying data can simplify the implementation of load balancing. For example, consider XML processing, which was considered as a typical application of tree skeletons [Ski97, NEM+07]. In using tree skeletons, DOM trees of unbounded degree are transformed into full binary trees. As a result, an input tree may be a list-like tree and be suited to load balancing based on segmented trees. However, the height of actual XML documents is small; e.g., XML documents of hundreds of height are unrealistic. List-like DOM trees are not worth considering. In fact, for the MapReduce-based implementation [EI12] of reduce over XML-like bracketed structures, load balancing to tame list-like trees was unnecessary as well as inefficient. Moreover, large-scale XML documents that constitute database usually have lists of the same kind of elements. It would be sufficient to parallelize computations on such lists simply as in list skeletons. The interpretation of substructures of data is thus of high importance for load balancing and practical implementations.

Tree skeletons deal solidly with programming *on* trees, where only the structure of trees matters. However, programming *with* trees, where trees are utilized for dealing with their underlying data, is truly desired. In the following chapters, we therefore deal with parallel programming with trees in a divide-and-conquer manner by considering the interpretation of trees and the underlying data primarily.

# Part II

# Syntax-Directed Programming

# Chapter 6

# Syntax-Directed Computation and Program Analysis

## 6.1 Motivation for Program Analysis

A syntax-directed computation is to calculate results from trees on the basis of the production rules of their grammars. Attribute grammars (AGs) [Knu68] are one of the most famous styles of syntax-directed computations. In fact, data-parallel skeletons is also a style of syntax-directed computations. For example, we first define the grammar (i.e., syntax) of join lists to formalize lists in the divide-and-conquer paradigm and then define reduce for each production rule on the basis of the interpretation of its production. That is, programming with data-parallel skeletons is a syntax-directed manner of programming, i.e., *syntax-directed programming*.

As mentioned in Chapter 5, it is important to consider the interpretation and underlying data of trees. The most typical target of syntax-directed computations is a computer program, where a syntax defines the abstract syntax tree (AST) of an input program. In fact, compiler frontends perform syntax-directed computations on ASTs and their implementation is the primary application of AGs [Wai90, WBGK10, WdMBK02, WKSB07, EH07]. We therefore focus on ASTs in the domain of programming languages.

Recall that data-parallel skeletons (especially, reduce) give an input tree an interpretation. In programming languages, to give a program an interpretation is generally called program analysis. For example, the work of compiler frontends, which are often defined by using AGs, is called semantic analysis. AST-based program analysis is very similar to data-parallel skeletons. We therefore consider that dealing with AST-based program analysis leads to structured parallel programming with trees.

In this part, we deal with AST-based program analysis as case studies on parallel programming with trees. We do not consider any dynamic analysis for two reasons. The first is that we usually perform it incrementally. The second is that it usually focuses on the executed part and ignores the non-executed part. These characteristics make the amount of work difficult to estimate. Such problems are inappropriate to parallel programming. Meanwhile, static analysis is usually exhaustive and deals with the whole program. Consequently, it tends to be expensive and thereby was accelerated through parallelization in many studies [NG13, MLBP12, AKNR12, PRMH11, RL11, MLMP10]. Static analysis is appropriate to the case studies on parallel programming. We therefore deal with static analysis.

## 6.2 High-Level Program Analysis

Program analyses after semantic analysis usually perform on control-flow graphs (CFGs) because they are independent of input languages. CFGs represent control flow directly and do not represent high-level control structures, which ASTs represent by definition. In this sense, AST-based analysis is higher-level than CFG-based analysis.

The analysis of high-level representations close to source languages is generally more precise and more

inexpensive than that of low-level ones close to target languages. Nevertheless, CFG-based approaches to program analysis are more popular than AST-based ones in optimizing compilers. This is because CFGs-based approaches deal with arbitrary control flow by definition. If source languages do not cause arbitrary control flow, compiler optimizations could destroy control structures and then lead to arbitrary control flow. In this sense, CFG-based approaches are easier to apply. This is why CFG-based approaches are more popular.

ASTs are more intuitive and easier to understand than CFGs. As demonstrated in the literature on AG-based compiler constructions [Wai90, WBGK10, WdMBK02, WKSB07, EH07], high-level approaches based on ASTs are useful for compiler constructions. Moreover, in editors integrated with compiler frontends such as Eclipse[1], AST-based approaches are quite natural. High-level program analysis based on ASTs is extensively desired. It is therefore worth migrating from de facto standard CFG-based implementation to AST-based implementation.

A main obstacle to migrating from CFGs to ASTs is the control flow that does not form a control structure, which compilers could introduce in the process of compilation. We can introduce such control flow into ASTs by using goto/label statements. It is, however, difficult to interpret the control flow derived from goto/label statements in a syntax-directed manner because it ignores the structures of ASTs. If we can deal with goto/label statements in a syntax-directed manner, we can replace CFG-based analysis with AST-based analysis. Consequently, we can enjoy the benefits from high-level program analysis as well as high-level compiler constructions. The primary technical issue in Chapters 7 and 8 is therefore how to tame with goto/label statements.

From the perspective of syntax-directed programming, how to tame goto/label statements corresponds to how to deal with irregular computations involved in computations with trees. To deal with such irregular part in a structured manner is a technical issue in structured parallel programming with trees. We therefore consider that to tame goto/label statements is very appropriate to a case study on parallel programming with trees.

[1] `https://eclipse.org/`

# Chapter 7

# Syntax-Directed Divide-and-Conquer Data-Flow Analysis

This chapter is self-contained and an extended version of our publication [SM14b].

## 7.1 Introduction

Data-flow analysis (DFA) is a classic and fundamental formalization in programming languages and particularly forms the foundation of compiler optimization. Many optimizations consist of a pair of analysis and transformation, and DFA often formulates the analysis part of an optimization and occupies the computational kernel of its optimization pass.

Nowadays, an input to DFA can be very large. For example, state-of-the-art optimizing compilers such as GCC and LLVM are equipped with link-time optimization (LTO), which is to reserve intermediate representations beside executables at compile time and then optimize the whole program at link time by using all reserved intermediate representations of linked executables. An input program of LTO is larger than the one of usual separate compilations. Furthermore, LTO promotes aggressive procedure inlining, which can incur an exponential blow up of input programs. In DFA for LTO, it is therefore desired that large-scale input programs can be dealt with effectively.

One promising approach to dealing with large-scale inputs is parallelization. Since parallel machines are widespread, well-parallelized DFA will benefit many users of LTO. A primary concern is the generation and assignment of parallel tasks. Concretely, load balancing with little overhead is important. Although load balancing is necessary to reduce parallel time, the load balancing itself could incur considerable overhead in processing large-scale inputs. For parallel DFA of large-scale input programs, the divide and conquer directly on input data structures without preprocessing is very much desired because this will result in the immediate generation of parallel finer-grained tasks in recursion.

A naive approach to the divide and conquer of DFA is procedure-level decomposition. In interprocedural as well as intraprocedural analysis, the analysis of each procedure is computationally almost independent of that of the others and therefore can be performed in parallel. This procedure-wise parallelization, however, can incur a poor load balancing in LTO with aggressive inlining. Aggressive inlining expands the main procedures sharply by substituting and eliminating many other procedures; consequently, it reduces the number of procedures and causes a size imbalance among procedures. To obtain better load balancing, the divide and conquer over a procedure is necessary.

DFA usually deals with a procedure in the form of a control-flow graph (CFG). Although there were some earlier studies on parallel DFA that developed divide-and-conquer methods on CFGs, these methods required an auxiliary tree structure [LRF95] or duplication of CFGs [KGS94] and therefore incur significant overhead. These drawbacks stem from the nature of CFGs. The loops and sharing of paths in CFGs make the divide and conquer of DFA difficult because they impose unstructured dependence on parts of the DFA. To resolve this dependence, some preprocessing is generally required. Therefore, DFA on CFGs is essentially difficult to divide and conquer.

In contrast to CFGs, abstract syntax trees (ASTs) are easy to divide and conquer owing to their recursive structures. If we can perform DFA on ASTs, the divide and conquer of DFA will be straightforward in a recursion on ASTs (i.e., a syntax-directed manner) and enable us to perform each DFA of independent AST subtrees in parallel. Rosen [Ros77] developed high-level data-flow analysis, a well-formed method of DFA on ASTs, but his method cannot deal with goto/label. Since goto/label causes control flow unrestricted to the structures of ASTs, it introduces into ASTs unstructured dependence similar to that of CFGs. Taming goto/label is therefore essential for general DFA.

To resolve this problem, we have developed a novel parallel syntax-directed method of general DFA that tames goto/label. The proposed method is built upon Tarjan's algebraic formalization [Tar81a] of DFA. First, our method summarizes the syntax-directed data flow in a bottom-up parallel sweep of a given AST, while detaching the goto-derived data flow and constructing a compact system of linear equations that represent it. Next, we obtain the summary of the goto-derived data flow by solving the system. Lastly, we merge the syntax-directed data flow with the goto-derived flow. Our method is particularly useful for programs containing few goto/label statements because the divide and conquer over a given AST is applied to the most part of DFA. We can assume such an input thanks to the popularity of structured programming. Furthermore, our method guarantees asymptotically linear speedup.

The following are our two major contributions:

- We have developed a novel syntax-directed divide-and-conquer parallel method of DFA based on Tarjan's formalization [Tar81a] (Section 7.3). The essence of our method is to detach the goto-derived data flow and calculate it afterward. Our method guarantees asymptotically linear speedup.
- We have also developed a practical technique to enhance our DFA method on the basis of the notion of interval analysis [AC76, Tar81b] (Section 7.4). Our technique will reduce the constant factors of our method for usual input programs.
- We have demonstrated the feasibility of our method experimentally through prototype implementations on a C compiler (Section 7.5). Our parallel prototype achieved a significant speedup and our sequential prototype achieved reasonable performance compared to the standard implementation.

## 7.2 Formalization of Data-Flow Analysis

DFA is to aggregate data-flow values over a given program [Kil73]. The domain of data-flow values is a join-semilattice $L$ whose join operator is $\sqcup$. Each program point has a transfer function over $L$. The result of DFA is defined as a join-over-all-paths (JOP) solution, namely, a sum of the data-flow values of all executable paths from the entry to the exit (or a target point) in a given program.

The proposed method is based upon Tarjan's formalization over a closed semiring [Tar81a]. This first formalizes an input program as the set of all executable paths represented by a *regular path*, which is a regular expression whose alphabet is the set of all program points $\Pi$. Then, DFA is defined as a homomorphism $h_R$ from a closed semiring $(R, |, \cdot, \varnothing, \epsilon)$ to another closed semiring $(F, \oplus, \otimes, \overline{0}, \overline{1})$. The former is for regular paths: $R$ is a set of regular paths, addition is the alternation $|$, and multiplication is the concatenation $\cdot$. The latter is for transfer functions: $F$ is the set of transfer functions, the addition $f_1 \oplus f_2 = \lambda x.f_1(x) \sqcup f_2(x)$, the multiplication[1] $f_1 \otimes f_2 = f_2 \circ f_1$, $\overline{0}$ is the zero element, $f \oplus \overline{0} = \overline{0} \oplus f = f$ and $f \otimes \overline{0} = \overline{0} \otimes f = \overline{0}$, and $\overline{1}$ is the multiplicative identity, $f \otimes \overline{1} = \overline{1} \otimes f = f$. Note that from the definition of a closed semiring, Kleene star $f*$ is defined as $f* = \bigoplus_{i=0}^{\infty} f^i$, where $f^0 = \overline{1}$ and $f^i = f^{i-1} \otimes f$. Giving $(F, \oplus, \otimes, \overline{0}, \overline{1})$ and a lift function $\tau : \Pi \to F$, we can characterize the homomorphism of DFA as

$$
\begin{aligned}
h_R(\epsilon) &= \overline{1}, \\
h_R(\pi) &= \tau(\pi), \quad \text{if } \pi \in \Pi, \\
h_R(r_1 \cdot r_2) &= h_R(r_1) \otimes h_R(r_2), \\
h_R(r_1 \mid r_2) &= h_R(r_1) \oplus h_R(r_2), \\
h_R(r*) &= h_R(r) * .
\end{aligned}
$$

[1] Here, we consider forward DFA. For backward DFA, $f_1 \otimes f_2 = f_1 \circ f_2$.

In this paper, we assume that $\varnothing$ is not an input of any DFA. Therefore, $\overline{0}$ can be left undefined and regarded as a special value that behaves as the zero element.

**Example of DFA** To give readers to a clearer understanding of Tarjan's formalization, here we describe the DFA of reaching definitions. A definition is a pair consisting of an LHS variable and an RHS expression. In this DFA, the domain of data-flow values is a set of definitions, i.e., a binary relation from variables to expressions. The join operation is the set union. The transfer function of an assignment statement $v \leftarrow e$ generates a definition $v \mapsto e$ and kills all other definitions of $v$ that can reach the assignment. Meanwhile, a simple expression $e$ without assignment has no effect on data flow. That is, $\tau$ is defined as

$$\begin{aligned} \tau(v \leftarrow e) &= \lambda X.\ \{v' \mapsto e \in X \mid v \neq v'\} \cup \{v \mapsto e\}, \\ \tau(e) &= \lambda X.\ X. \end{aligned}$$

To define a closed semiring, a general form of transfer functions is necessary. Letting $V$ be a set of variables and $D$ be a set of definitions, we can define it as

$$f(V, D) = \lambda X.\ \{v \mapsto e \in X \mid v \notin V\} \cup D.$$

By using this $f$, we can define $\tau$ and a closed semiring $(F, \oplus, \otimes, \overline{0}, \overline{1})$ as

$$\begin{aligned} \tau(v \leftarrow e) &= f(\{v\}, \{v \mapsto e\}), \\ \tau(e) &= f(\varnothing, \varnothing), \\ \overline{1} &= f(\varnothing, \varnothing), \\ f(V_1, D_1) \oplus f(V_2, D_2) &= f(V_1 \cap V_2, D_1 \cup D_2), \\ f(V_1, D_1) \otimes f(V_2, D_2) &= f(V_1 \cup V_2, \{v \mapsto e \in D_1 \mid v \notin V_2\} \cup D_2), \\ f(V, D)* &= 1 \oplus f(V, D). \end{aligned}$$

As seen in the $h_R$ above, Tarjan's approach calculates a *summary*[2], namely a transfer function for a program fragment, rather than data-flow values. By applying the summary from an entry to an exit to a given initial data-flow value, we obtain its JOP solution. This formalization can deal with monotone DFA. Refer to [Tar81a, MR90] for a detailed discussion.

For optimizations, compilers often use JOP solutions from an entry to every point, i.e., all-points JOP solutions. Although the homomorphism above does not calculate the summaries for all-points JOP solutions, it is easy to calculate them. We can obtain a set of summaries from an entry to all points by accumulating summaries over a regular path, similarly to calculating a prefix sum. We call this an all-points summary. By applying each element of an all-points summary to an initial value, we obtain all-points JOP solutions.

In Tarjan's formalization, the primary concern on algorithms is how to construct the regular path of an input program. Tarjan [Tar81b] developed a sophisticated algorithm for extracting a regular path from a CFG. However, if an input program is goto-free, namely, in the while language (Fig. 7.1), we can immediately obtain its regular path representation. This is trivial but notable. Thus, DFA for the while language is performed in a syntax-directed manner as follows:

$$\begin{aligned} h(\textbf{pass}) &= \overline{1}, \\ h(v \leftarrow e) &= \tau(v \leftarrow e), \\ h(s_1\ s_2) &= h(s_1) \otimes h(s_2), \\ h(\textbf{if}\ (e)\ \{s_1\}\ \textbf{else}\ \{s_2\}) &= \tau(e) \otimes (h(s_1) \oplus h(s_2)), \\ h(\textbf{while}\ (e)\ \{s\}) &= \tau(e) \otimes (h(s) \otimes \tau(e))*. \end{aligned}$$

Here, $h$ calculates the summary of a given program fragment. Throughout this paper, we identify a program fragment given to $\tau$ with its program point; thus, $\tau$ takes a program fragment and yields a transfer function.

[2] A procedure summary, which is the transfer function of the whole of a procedure, is used extensively for interprocedural analysis [SP81].

$$
\begin{array}{rcll}
P & ::= & s & \text{(Program)} \\
s & ::= & \textbf{pass} \mid v \leftarrow e \mid s_1\ s_2 \mid \textbf{if}\ (e)\ \{s_1\}\ \textbf{else}\ \{s_2\} \mid \textbf{while}\ (e)\ \{s\} & \text{(Statement)}
\end{array}
$$

Figure 7.1: Syntax of the while language. $v$ and $e$ are respectively the metavariables over variables and expressions; **pass** denotes an empty statement.

**Example** We explain the syntax-directed DFA based on Tarjan's formalization by using the following example program:

$$
\begin{aligned}
x \leftarrow a\ &\textbf{if}\ (x < 0)\ \{x \leftarrow 0\} \\
&\textbf{else}\ \{\textbf{while}\ (x < 10)\ \{x \leftarrow x + a\}\}.
\end{aligned}
$$

By applying $h$, we calculate the summary of the above program as

$$
\begin{aligned}
\tau(x \leftarrow a) \otimes \tau(x < 0) \otimes (&\tau(x \leftarrow 0) \\
&\oplus (\tau(x < 10) \otimes (\tau(x \leftarrow x + a) \otimes \tau(x < 10))*)).
\end{aligned}
$$

By using the closed semiring of reaching definitions, we can reduce it to

$$f(\{x\}, \{x \mapsto 0,\ x \mapsto x + a\}).$$

Because the initial data-flow value of reaching definitions is $\varnothing$, the JOP solution at the exit is

$$\{x \mapsto 0,\ x \mapsto x + a\}.$$

We can also construct all-points summaries in a syntax-directed manner. An example of such construction is described in Section 7.3.3.

## 7.3 Syntax-Directed Parallel DFA Algorithm

For goto-free programs, the divide and conquer of DFA is immediate from a syntax-directed computation, and its parallelization is therefore straightforward. Syntax-directed jumps (i.e., jumps to ancestors on ASTs) such as break/continue can be dealt with by using Rosen's method [Ros77] in a syntax-directed manner. Non-syntax-directed (i.e., nonstructural) jumps caused by goto/label, however, require a special attention. In the following, our target language is the while language with goto/label. Letting $l$ be a metavariable over labels, we introduce a goto statement **goto** $l$ and a label statement $l$:.

The main idea of the proposed method is to discriminate between syntax-directed (i.e., structural) data flow and goto-derived (i.e., nonstructural) data flow. Our method consists of two phases: first, it constructs a summary of structural data flow while detaching nonstructural data flow from an input AST in a syntax-directed manner, and second, it calculates only nonstructural data flow from the obtained summary. After that, we obtain JOP solutions.

In terms of parallelization, the first phase is straightforward from a syntax-directed computation. This is the main benefit of our method. We do not have to parallelize the second phase. The size of a summary obtained in the first phase is quadratic to the number of labels. Because of the popularity of structured programming, we can suppose that labels are few; that is, we assume nonstructural flow to be an exceptional irregularity in an input. The second phase would be cheap and not worth parallelizing. In the rest of this section, we describe the algorithms of both phases and the extension to interprocedural analysis.

### 7.3.1 Syntax-Directed Construction of Summaries

It is nontrivial to represent a program that contains goto/label by a single regular path. For example, consider the following program:

$$\textbf{while}\ (\underline{x < 10_1})\ \{l\text{: } \underline{x \leftarrow x + 1_2}\}\ \textbf{if}\ (\underline{x > 20_3})\ \{\textbf{goto}\ l\}\ \textbf{else}\ \{\underline{\textbf{pass}_4}\},$$

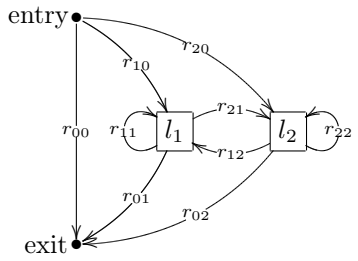



Figure 7.2: Regular paths in a program containing labels $l_1$ and $l_2$. $r_{00}$ is the regular path from the entry to the exit, $r_{0i}$ is the regular path from $l_i$: to the exit, $r_{i0}$ is the regular path from the entry to **goto** $l_i$, and $r_{ij}$ is the regular path from $l_j$: to **goto** $l_i$.

where a suffix to an underlined part denotes its program point. We cannot construct a Kleene closure only from the while statement above unlike the goto-free case because regular paths containing jumps to $l$: are unknown. We, however, can decompose by interpreting $l$: as another entry and **goto** $l$ as another exit, the above program into four goto-free regular paths: $1 \cdot (2 \cdot 1)* \cdot 3 \cdot 4$ (from the entry to the exit), $1 \cdot (2 \cdot 1)* \cdot 3$ (from the entry to **goto** $l$), $2 \cdot 1 \cdot (2 \cdot 1)* \cdot 3 \cdot 4$ (from $l$: to the exit), and $2 \cdot 1 \cdot (2 \cdot 1)* \cdot 3$ (from $l$: to **goto** $l$). These are immediately obtained from the AST. In the case of two labels $l_1$ and $l_2$, we can generally consider nine regular paths as illustrated in Fig. 7.2, where all goto-derived jumps to $l_i$ are encapsulated in the box labeled by $l_i$. This decomposition enables us to postpone interpreting goto-derived flow. This is the key idea of our method.

On the basis of this idea, we define a structured summary by a set of transfer functions. Let $\{l_1, \ldots, l_k\}$ be the set of labels and $a_{ij} = h_R(r_{ij})$, where $r_{ij}$ is the goto-free regular path from $l_j$: (or the entry, if $j = 0$) to **goto** $l_i$ (or the exit, if $i = 0$); then, a structured summary is the following system of linear equations:

$$\left\{\begin{array}{r} out = a_{00} \oplus (l_1 \otimes a_{01}) \oplus \cdots \oplus (l_k \otimes a_{0k}), \\ l_1 = a_{10} \oplus (l_1 \otimes a_{11}) \oplus \cdots \oplus (l_k \otimes a_{1k}), \\ \vdots \\ l_k = a_{k0} \oplus (l_1 \otimes a_{k1}) \oplus \cdots \oplus (l_k \otimes a_{kk}), \end{array}\right\}$$

where *out* denotes the data flow that goes out from the exit and $l_i$ denotes nonstructural data flow via the label $l_i$; specifically, $l_i$ denotes an outflow in the LHS and an inflow in the RHS. In the rest of this paper, we omit any equation whose RHS is $\overline{0}$. We can represent the system above by a coefficient matrix,

$$\begin{pmatrix} out \\ \boldsymbol{x} \end{pmatrix} = \begin{pmatrix} \overline{1} \\ \boldsymbol{x} \end{pmatrix}^T A^T, \quad \text{where } \boldsymbol{x} = \begin{pmatrix} l_1 \\ \vdots \\ l_k \end{pmatrix}, \ A = \begin{pmatrix} a_{00} & \cdots & a_{0k} \\ \vdots & \ddots & \vdots \\ a_{k0} & \cdots & a_{kk} \end{pmatrix}.$$

The matrix multiplication here is defined by using $\oplus$ and $\otimes$ respectively as the scalar addition and the scalar multiplication. Unless otherwise noted, matrix operations are generalized over a semiring. For a structured summary, we intentionally confuse the system of linear equations with its coefficient matrix $A$.

We define each of the addition, multiplication, and Kleene star over structured summaries as a matrix operation.

The addition is used to merge two independent summaries, such as those of two branches of a conditional statement. It is easy to see that the conventional matrix addition suffices for this purpose; consider the edge-wise union on Fig. 7.2. In the rest of this paper, we overload $\oplus$ for the matrix addition.

The multiplication is used to connect the summaries of two consecutive statements. This necessitates a little consideration. For example, consider the concatenation of two copies of the regular paths in Fig. 7.2, as illustrated in the left side of Fig. 7.3. Although two boxes labeled by $l_i$ exist there, both encapsulate the same kind of control flow. We can therefore contract regular paths by merging both

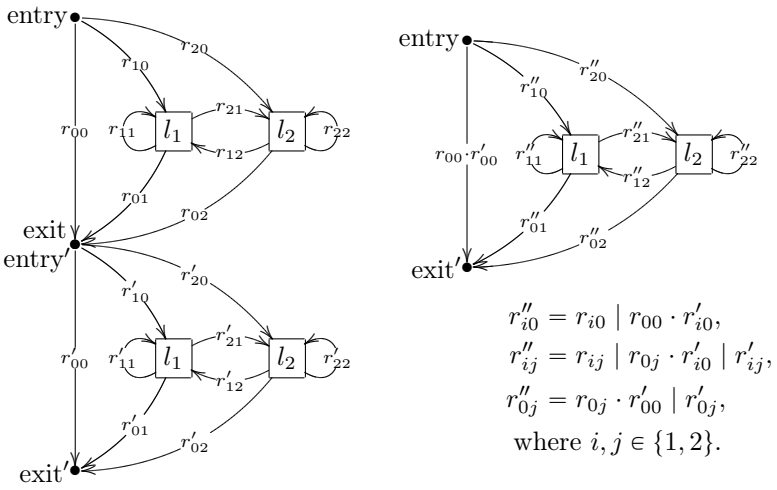



Figure 7.3: Concatenation of two copies of the regular paths in Fig. 7.2, where the left side is the connected view and the right side is the contracted view.

boxes, as illustrated in the right side of Fig. 7.3. This contraction of regular paths leads to the following definition of the multiplication $\odot$:

$$(c_{ij})_{0 \leqslant i,j \leqslant k} = (a_{ij})_{0 \leqslant i,j \leqslant k} \odot (b_{ij})_{0 \leqslant i,j \leqslant k}$$
$$s.t.\ c_{ij} = \begin{cases} a_{00} \otimes b_{00} & (i = j = 0), \\ (a_{0j} \otimes b_{00}) \oplus b_{0j} & (i = 0 \wedge j \neq 0), \\ (a_{00} \otimes b_{i0}) \oplus a_{i0} & (i \neq 0 \wedge j = 0), \\ (a_{0j} \otimes b_{i0}) \oplus a_{ij} \oplus b_{ij} & (i \neq 0 \wedge j \neq 0). \end{cases}$$

Note that $\odot$ is associative and its identity is $\{out = \overline{1}\}$.

By using the addition $\oplus$ and the multiplication $\odot$, we can define the Kleene star in the standard way. However, owing to the idempotence of $\oplus$, we can provide the following equivalent but simpler definition:

$$(a_{ij})_{0 \leqslant i,j \leqslant k}{*} = (a'_{ij})_{0 \leqslant i,j \leqslant k}$$
$$s.t.\ a'_{ij} = \begin{cases} a_{00}{*} & (i = j = 0), \\ a_{0j} \otimes a_{00}{*} & (i = 0 \wedge j \neq 0), \\ a_{00}{*} \otimes a_{i0} & (i \neq 0 \wedge j = 0), \\ (a_{0j} \otimes a_{00}{*} \otimes a_{i0}) \oplus a_{ij} & (i \neq 0 \wedge j \neq 0). \end{cases}$$

Now we are ready to define $h_{\mathcal{C}}$, which calculates a structured summary from a given AST.

$$\begin{aligned} h_{\mathcal{C}}(e) &= \{out = h(e)\}, \\ h_{\mathcal{C}}(s) &= \{out = h(s)\}, \\ h_{\mathcal{C}}(l_i\text{:}) &= \{out = \overline{1} \oplus l_i\}, \\ h_{\mathcal{C}}(\textbf{goto}\ l_i) &= \{l_i = \overline{1}\}, \\ h_{\mathcal{C}}(s'_1\ s'_2) &= h_{\mathcal{C}}(s'_1) \odot h_{\mathcal{C}}(s'_2), \\ h_{\mathcal{C}}(\textbf{if}\ (e)\ \{s'_1\}\ \textbf{else}\ \{s'_2\}) &= h_{\mathcal{C}}(e) \odot (h_{\mathcal{C}}(s'_1) \oplus h_{\mathcal{C}}(s'_2)), \\ h_{\mathcal{C}}(\textbf{while}\ (e)\ \{s'\}) &= h_{\mathcal{C}}(e) \odot (h_{\mathcal{C}}(s') \odot h_{\mathcal{C}}(e))*, \end{aligned}$$

where $s$ denotes a metavariable over statements containing no goto/label statement and $s'$ denotes one over statements containing any goto/label statement.

**Example** We here consider, as an example input, the following program:

$$\begin{aligned}&\textbf{if } (i > n)\ \{l_1\colon x \leftarrow 1\}\ \textbf{else}\ \{\textbf{while } (i < n)\ \{i \leftarrow 2\}\ l_2\colon x \leftarrow i\}\\&\textbf{if } (x = n)\ \{i \leftarrow x\}\ \textbf{else}\ \{\textbf{goto } l_1\}.\end{aligned}$$

By applying $h_{\mathcal{C}}$, we calculate a structured summary of the above program as

$$\begin{aligned}&A_1 \odot ((\{out = \overline{1} \oplus l_1\} \odot A_2) \oplus (A_3 \odot \{out = \overline{1} \oplus l_2\} \odot A_4)) \odot A_5 \odot (A_6 \oplus \{l_1 = \overline{1}\}),\\&\text{where } A_1 = \{out = \tau(i > n)\},\\&\qquad A_2 = \{out = \tau(x \leftarrow 1)\},\\&\qquad A_3 = \{out = \tau(i < n) \otimes (\tau(i \leftarrow 2) \otimes \tau(i < n))*\},\\&\qquad A_4 = \{out = \tau(x \leftarrow i)\},\\&\qquad A_5 = \{out = \tau(x = n)\},\\&\qquad A_6 = \{out = \tau(i \leftarrow x)\}.\end{aligned}$$

By reducing matrix operations, we obtain

$$\begin{aligned}&\{out = a'_1 \oplus (l_1 \otimes a'_2) \oplus (l_2 \otimes a'_3),\ l_1 = a'_4 \oplus (l_1 \otimes a'_5),\}\\&\text{where } a'_1 = a_1 \otimes (a_2 \oplus (a_3 \otimes a_4)) \otimes a_5 \otimes a_6,\\&\qquad a'_2 = a_2 \otimes a_5 \otimes a_6,\\&\qquad a'_3 = a_4 \otimes a_5,\\&\qquad a'_4 = a_1 \otimes (a_2 \oplus (a_3 \otimes a_4)) \otimes a_5,\ a'_5 = a_2 \otimes a_5\\&\qquad a_1 = \tau(i > n),\\&\qquad a_2 = \tau(x \leftarrow 1),\\&\qquad a_3 = \tau(i < n) \otimes (\tau(i \leftarrow 2) \otimes \tau(i < n))*,\\&\qquad a_4 = \tau(x \leftarrow i),\ a_5 = \tau(x = n),\ a_6 = \tau(i \leftarrow x).\end{aligned}$$

This result exemplifies the notion of a structured summary: $a'_1$ denotes the data flow from the entry to the exit, $a'_2$ denotes that from the $l_1$: to the exit, $a'_3$ denotes that from $l_2$: to the exit, $a'_4$ denotes that from the entry to the **goto** $l_1$, and $a'_5$ denotes that from $l_1$: to the **goto** $l_1$. None of these take any nonstructural data flow into account, but the whole system contains the nonstructural data flow of the program. For example, $l_1 = a'_4 \oplus (l_1 \otimes a'_5)$ denotes that the nonstructural data flow via $l_1$ is $a'_4 \otimes a'_5*$. We can therefore calculate the nonstructural data flow from this structured summary. Finally, we obtain the value of *out*.

**Parallel Complexity** We can parallelize $h_{\mathcal{C}}$ immediately in a divide-and-conquer manner because $h_{\mathcal{C}}$ can fork for each child at any internal node of an input AST. For the parallel time complexity of $h_{\mathcal{C}}$, the associativity of $\odot$ is important. We can flatten the nesting of statement sequencing, i.e., convert a nesting $((s_1\ s_2)\ s_3)$ into a sequence $(s_1\ s_2\ s_3)$, because it guarantees both results to be equivalent. Moreover, it enables us to perform parallel reduction for a sequence of statements. The number of parallel recursive steps of $h_{\mathcal{C}}$ therefore is bounded by the maximum if/while nesting $d$ in an input AST. Let $k$ be the number of labels, $N$ be the number of the nodes in an input AST, $P$ be the number of processors, and $b$ be the maximum length of a sequence of statements. The parallel time complexity of $h_{\mathcal{C}}$ is the following:

$$\mathrm{O}(k^2(N/P + d \lg \min(b, P))),$$

where we assume closed-semiring operations to be constant-time. This $\lg \min(b, P)$ factor is derived from the parallel reduction of a sequence of statements and is practically negligible. The $k^2$ factor represents the cost of matrix operations. Note that for an AST containing no label statement, this factor will be $k$, and for one containing no goto/label statement, it will be a constant. If $N/P > d \lg \min(b, P)$, $h_{\mathcal{C}}$ guarantees asymptotically linear speedup.

### 7.3.2 Calculating Join-Over-All-Paths Solutions

To obtain a JOP solution, we have to solve the nonstructural data flow whose calculation has been postponed, i.e., to determine the value of $\boldsymbol{x}$ in a structured summary. As seen in Fig. 7.2, a structured summary can be regarded as a collapsed CFG. We can therefore apply existing methods on CFGs to solve that. The simplest one is Gaussian elimination [RP86]. Although it is cubic-time, it is sufficient to solve the nonstructural data flow. Assuming closed-semiring operations to be constant-time, it costs only $\mathrm{O}(k^3)$ because of the size of a structured summary as a CFG. This cost is asymptotically negligible compared to the parallel cost of $h_{\mathcal{C}}$ if $N/P + d \lg \min(b, P) > k$. Therefore, the part to solve nonstructural data flow is not worth sophisticating and/or parallelizing.

Once the value of $\boldsymbol{x}$ in a structured summary is obtained, we can determine the value of *out* in $\mathrm{O}(k)$ time. By applying a initial value to *out*, we obtain the JOP solution of a given program. It is usually constant-time.

### 7.3.3 Construction of All-Points Summaries

We can compute all-points JOP solutions from an all-points summary in embarrassingly parallel because each application of its elements to an initial value is independent. We can construct all-points summaries by using tree accumulation.

The tree accumulation to construct an all-points summary consists of two phases. The first is the same as $h_{\mathcal{C}}$ except for leaving intermediate results at each node in a given AST. The second is a top-down sweep of the AST decorated with intermediate results. In this top-down sweep, we perform the parallel prefix-sum operation with $\odot$ on every sequence of statements and update summaries that decorate each node of the AST. The resultant AST decorated with structured summaries is an all-point summary. Note that $\odot$ used in the second phase has only to calculate the uppermost row vector and the leftmost column vector in a resultant matrix because only the equation of *out* in every element of an all-points summary is used for yielding all-points JOP solutions. The second phase is cheaper than the first one. Therefore, the time complexity of constructing an all-points summary is the same as that of $h_{\mathcal{C}}$.

**Example** The above algorithm for constructing all-points summaries is, in fact, applicable to both $h$ and $h_{\mathcal{C}}$. The difference between them is only on primitive operations: scalar ones (e.g., $\otimes$) used for $h$ and matrix ones (e.g., $\odot$) used for $h_{\mathcal{C}}$. For simplicity, we describe here the construction of an all-points summary regarding $h$. We consider the following goto-free program:

$$\textbf{if}\ (e_1)\ \{s_1\ s_2\ s_3\}\ \textbf{else}\ \{\textbf{while}\ (e_2)\ \{s_4\ s_5\}\ s_6\}.$$

We reserve part of the above program as metavariables to concentrate a recursive step. After the first phase of bottom-up tree accumulation, we obtain

$$\begin{aligned}&\textbf{if}\ (f_1)\ \{f_2\ f_3\ f_4\}\ \textbf{else}\ \{f_6\ f_7\},\\ &\text{where } f_1 = h(e_1),\ f_2 = h(s_1),\ f_3 = h(s_2),\ f_4 = h(s_3),\\ &\qquad f_5 = h(e_2),\ f_6 = h(\textbf{while}\ (e_2)\ \{s_4\ s_5\}),\ f_7 = h(s_6).\end{aligned}$$

Tree accumulation also brings us the summaries of all next-level statements; e.g., we have already had

$$\begin{aligned}&\textbf{while}\ (f'_1)\ \{f'_2\ f'_3\},\\ &\text{where } f'_1 = h(e_2),\ f'_2 = h(s_4),\ f'_3 = h(s_5).\end{aligned}$$

In the second phase, we calculate a top-down prefix sum at each nesting level. The following is the result for the outermost if statement:

$$\begin{aligned}&\textbf{if}\ (f_1)\ \{(f_1 \otimes f_2)\ (f_1 \otimes f_2 \otimes f_3)\ (f_1 \otimes f_2 \otimes f_3 \otimes f_4)\}\\ &\textbf{else}\ \{(f_1 \otimes f_6)\ (f_1 \otimes f_6 \otimes f_7)\}.\end{aligned}$$

We then recurse on next-level statements: $s_1$, $s_2$, $s_3$, **while** $(e_2)$ $\{s_4\ s_5\}$, and $s_6$. Since we have already had all these summaries in the first phase, we are ready to recurse on them. After recursions on statements at all levels, the resultant AST becomes an all-points summary.

### 7.3.4 Interprocedural Analysis

Tarjan's formalization deals essentially with intraprocedural DFA. However, it can be extended to calculate procedure summaries and is therefore useful even for interprocedural DFA. In fact, our method can deal effectively with context-insensitive interprocedural DFA.

We now consider a program $P$ to be a set of top-level procedures. Let $p$ be a metavariable over procedure names. The syntax of a procedure with a body statement $s$ is $p()\{s\}$. The procedure call $p()$ and **return** are introduced to $s$. For simplicity, we assume that none of the procedures take arguments or return values. Argument passing and value returning may be implemented by using global variables. For convenience, $p_c$ refers to the current procedure at a point.

Since the information of call sites is neglected in context-insensitive DFA, we can interpret call/return simply as goto/label. We extend $h_{\mathcal{C}}$ as follows:

$$\begin{aligned} h_{\mathcal{C}}(P) &= \bigoplus_{p \in P} h_{\mathcal{C}}(p()\{s\}), \\ h_{\mathcal{C}}(p()\{s\}) &= \{\} \odot h_{\mathcal{C}}(p_{\text{call}}\text{: } s \textbf{ goto } p_{\text{ret}}) \odot \{\}, \\ h_{\mathcal{C}}(p()) &= h_{\mathcal{C}}(\textbf{goto } p_{\text{call}}\ p_{\text{ret}}\text{:}), \\ h_{\mathcal{C}}(\textbf{return}) &= h_{\mathcal{C}}(\textbf{goto } p_{\text{ret}}), \quad \text{where } p = p_c. \end{aligned}$$

Note that the null system $\{\}$ denotes no control flow. The same call-site label $p_{\text{ret}}$ may be attached to many program points. In such cases, we interpret goto as a nondeterministic jump to one of the corresponding label statements, where we require no change in $h_{\mathcal{C}}$. The rest of the DFA process, including the constructions of all-points summaries, is the same as the intraprocedural case.

In contrast, our method is less effective for context-sensitive interprocedural DFA because context sensitivity prevents us from factoring out the data flow of calls as a compact linear system. When using our method, the first choice to obtain context sensitivity is procedure inlining. Intraprocedural DFA with inlining is generally more precise than context-sensitive DFA. Furthermore, we usually require code replication similar to inlining for generating context-sensitively optimized code, and in this sense, inlining is essential for utilizing context sensitivity in compiler optimization. Although the drawback of inlining is the expansion of procedure sizes, it is tractable in our method by using divide-and-conquer parallelization. Our method is synergistic with inlining, and aggressive inlining followed by context-insensitive DFA is therefore both appropriate and sufficient.

## 7.4 Elimination of Labels

Our method assumes that a source program does not contain a lot of labels because structured programming has been already widespread. This assumption is crucial for our method in the time and space complexities. Unfortunately, aggressive inlining at a high optimization level multiplies the occurrences of goto/label. In this section, we describe techniques to eliminate labels.

The goto/label statements introduced in inlining has a locality in an AST since their jumps is originally closed in a procedure. We can eliminate the equations of nonstructural data flow via *closed* labels in a structured summary in collapsing, i.e., in an on-the-fly manner.

A label $l$ is closed in a structured summary $A$ iff the fragment that $A$ summarizes contains $l$: and all occurrences of **goto** $l$. An equation of nonstructural data flow via a closed label in a structured summary is eliminable. *On-the-fly elimination* is to apply an elimination method to an eliminable part of a structured summary in collapsing. To eliminate a part of a system of data-flow equations is very common in elimination methods [RP86]. On-the-fly elimination is only its instance.

We can check the closedness of labels in a simple way: reference counting. We first count the occurrences of each label on a syntactic summary and store the maximum count of each label. Then, in collapsing, we count the occurrences of each label while computing summaries. If the current count of a label reaches its stored maximum, we can apply on-the-fly elimination to it. This reference counting is lightweight and incurs only negligible overhead. As a side effect of this reference counting, it enables us to neglect the labels to which no goto statement jumps.

Another elegant way to check the closedness of labels is to introduce block scope into labels. If a label go out of scope, we can eliminate it. This approach is simple but very effective for labels that aggressive inlining multiplies. It is also effective for labels that compiler frontends often generate such as in the translation of control constructs such as break/continue and logical operators with short-circuit evaluation.

## 7.5 Experiments

We conducted experiments to demonstrate the feasibility and scalability of our algorithm. Note that our aim is not to evaluate our analyzer implementation.

### 7.5.1 Prototype Implementations

We implemented our method for the DFA of reaching definitions, which is the most standard and lightweight example of DFA. Because a lightweight computation to a large-scale input is sensitive to the overhead of load balancing, the DFA of reaching definitions is appropriate for demonstrating the scalability of our method. For simplicity, we did not implement on-the-fly elimination. Our implementations built upon COINS[3], a C compiler in Java. We implemented $h_{\mathcal{C}}$ as a simple visitor on an AST. We used a dense matrix for Gaussian elimination to solve nonstructural data flow. We made extensive use of `java.util.HashMap` for the implementation of the closed semiring of reaching definitions. We call our sequential prototype `seq` and the parallel one `par`. This parallelization was very simple; we simply used Java 7 Fork/Join framework for the visitor of $h_{\mathcal{C}}$. We forked a visitor for each compound statement in a sequence of statements while summarizing segments of atom statements. At the end of a sequence of statements, we waited for all forked visitors one by one and then calculated the summary of the sequence.

As the reference implementation of DFA, we implemented wordwise analysis [KD94], which is an efficient iterative method for solving the most common DFA, a.k.a. the bit-vector framework. We call this implementation `bvf`. Since this method uses a wordwise worklist, we implemented a sparse wordwise bit-vector. We used a LIFO queue as the worklist. We constructed a CFG of basic blocks, and then numbered definitions on the AST through the CFG. After that, we initialized gen/kill sets of each node and performed the iterative method.

Note that `bvf` by definition calculated all-points JOP solutions, while our prototypes `seq` and `par` calculated a procedure summary. The comparison of their absolute performance is therefore unfair. Because this difference on results stems from the difference on style between our method and the iterative method, a truly fair comparison is difficult. However, since the asymptotic time complexity of constructing an all-points summary is the same as $h_{\mathcal{C}}$ (see Section 7.3.3), in terms of asymptotic performance, `seq` and `par` are comparable to `bvf`.

### 7.5.2 Experimental Setup

We generated a large-scale input program normalized in the while language with goto/label statements by using a biased random generation. We set the maximum depth to about 128, the length of block statements to a random number between 1 and 8. An about half of if statements had empty else branches. Each goto statement was guarded by a simple if statement to avoid dead code. An about half of assignments defined new variables. The generated AST had about 1,000,000 statements where the number of goto and label statements were 96 and 20. We used this unrealistically large-scale program for a benchmark to observe asymptotic behaviors of our method. We call it `rand`.

To obtain a realistic large program, we used procedure inlining of recursive programs. As an example recursive program, we selected the Lua 5.2.3 parser[4], which is known to be written in clean C. After normalization, we applied inlining iteratively to the entry function. We stopped the recursion of inlining at the seventh level. The resultant entry function consisted of about 12,000 statements, where 51 pairs of goto/label statements existed. We call it `inl`.

[3] `http://coins-compiler.sourceforge.jp/`
[4] `http://www.lua.org/ftp/lua-5.2.3.tar.gz`

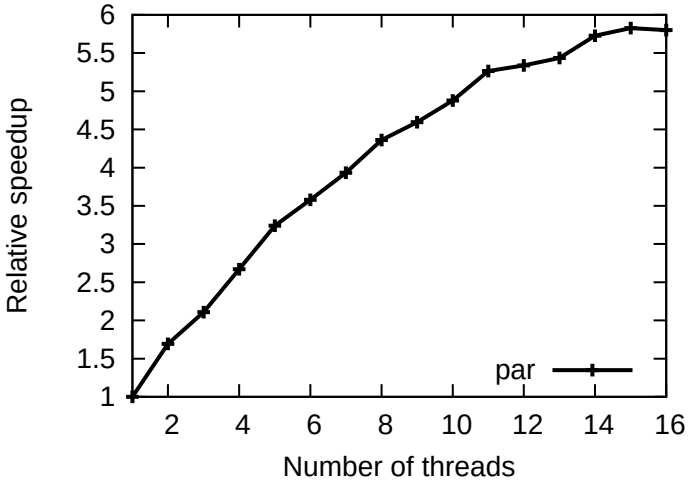



Figure 7.4: Relative speedup of `par` given `rand`.

Table 7.1: Breakdown of execution time of `seq`. "Elim" means Gaussian elimination on a structured summary.

| Phase | | $h_{\mathcal{C}}$ | Elim | Total |
|---|---|---|---|---|
| Time (ms) | `rand` | 810 | 1 | 811 |
| | `inl` | 13 | 1 | 14 |

We used a server equipped with four Opteron 6380 (16 cores, 2.50 GHz) processors and 128 GB of DDR3-1600 memory running OpenJDK (64-bit Server VM) version 1.7.0_55. We executed each analyzer 20 times for the same AST in memory. To minimize the effect of GC and VM issues, we discarded outliers and considered the median of the remainder as the result.

### 7.5.3 Experimental Results

The relative speedup of `par` given `rand`, shown in Fig. 7.4, had a significant scalability up to 15 threads. The relative speedup with 15 threads was 5.82x (while the speedup compared to `bvf` was 5.00x). A careful control of task granularity was not required. We also tested a granularity-controlled prototype but did not observe any performance gain. Only the divide and conquer of our method was sufficient to obtain a significant speedup. Our method was ready to parallelize and demonstrated that the divide and conquer on input data structures is crucial. The speedup curve in Fig. 7.4 demonstrates the asymptotically linear speedup of our method and exemplifies Amdahl's law.

Tables 7.1 and 7.2 respectively show the breakdowns of the execution time of `seq` and `bvf` given `rand` and `inl`. For `inl`, both `seq` and `bvf` were sufficiently fast. For `rand`, `seq` was significantly faster than `bvf`, but the direct comparison of both is inappropriate as mentioned earlier. What we can justify from these results is that our method is not algorithmically slower than the iterative method. It is notable that the Elim phase in our method incurred no overhead as expected. Therefore, our method is both feasible and useful if label statements in a given program are few, specifically less than about 50.

Table 7.2: Breakdown of execution time of `bvf`. "Cons" means CFG construction, "Ndef" means numbering definitions, "Init" means initializing gen/kill sets, and "Iter" means the iterative method.

| Phase | | Cons | Ndef | Init | Iter | Total |
|---|---|---|---|---|---|---|
| Time (ms) | `rand` | 431 | 565 | 971 | 334 | 2301 |
| | `inl` | 6 | 3 | 6 | 3 | 18 |

## 7.6 Related Work

Rosen [Ros77, Ros80] proposed the concept and method of high-level DFA. His method performs DFA on a high-level CFG that captures syntactic nesting, by calculating bit-vector equations for each level of statements similarly to interval analysis [AC76] but in a much finer-grained manner. Although Rosen dealt with break/continue, he did not with goto/label. The equations constructed in his method correspond to structured summaries containing only the leftmost column vector. His method can potentially deal with goto statements (i.e., jump-out) but not with label statements (i.e., jump-in). Mintz et al. [MFS79] implemented Rosen's method integrated with a CFG-based method to deal with goto/label. Their method processes ASTs containing no jump-in similarly to Rosen's. For ASTs containing a jump-in from the outside, it abandons the idea of calculating equations and instead constructs a CFG. For ASTs containing no jump-in from the outside but with a jump(s) between its components, by applying a CFG-based method to the CFG derived from the AST, the CFG is reduced to equations. The primary difference between our method and theirs is how the jump-in is handled. In our method, we detach the data flow of every jump-in completely from an input AST and summarize it into a structured summary. As a result, our method performs syntax-directed computation more thoroughly (e.g., even for context-insensitive interprocedural DFA) than theirs. This trait is quite advantageous in terms of divide-and-conquer parallelization.

In previous studies on parallelizing DFA [LRF95, KGS94], load balancing was the primary concern. Lee et al. [LRF95] improved the parallelization of interval analysis [AC76], where CFGs are recursively decomposed into substructures called intervals. In interval analysis, exclusive intervals can be processed in parallel, but the size of each interval, i.e., the granularity of parallel tasks is diverse. Lee et al. divided a CFG into controlled-size regions instead of intervals for load balancing and used an auxiliary tree structure to manage the parallelism among regions. Region decomposition itself is a sequential task. Kramer et al. [KGS94] utilized the parallel prefix-sum operation with $\otimes$ for each path of a CFG. Their method unwinds loops to convert a CFG into a directed acyclic graph. This degrades the generality of DFA. To make matters worse, their method expands the sharing of paths in a given CFG. This causes the asymptotic cost of DFA to blow up exponentially[5]. Our method is a simpler and cheaper way of divide-and-conquer parallelization, and furthermore guarantees asymptotically linear speedup.

Many studies on accelerating static analysis [NG13, MLBP12, AKNR12, PRMH11, RL11, MLMP10] parallelized fixed-point iterations. Multithreading with worklists [NG13, AKNR12, RL11] worked well for expected inputs in practical usage, but this imposes concurrency issues such as mutual exclusion for worklists, termination detection, deadlock/livelock, and the fairness of underlying schedulers. Parallel implementations specialized for GPUs [MLBP12, PRMH11] achieved high performance experimentally, but these techniques are very hardware-specific. Speculative parallelization [MLMP10] was feasible, but it complicates runtime behaviors. None of these approaches guarantee asymptotic speedup.

## 7.7 Conclusion

We have presented a novel syntax-directed parallel method of DFA that tames goto/label, and also experimentally demonstrated its feasibility and scalability.

There are two directions for future work. One is to implement our method more seriously by tying it to compiler optimizations and then to evaluate it practically. We expect that our method will simplify the construction of optimizing compilers. The other is to apply our method to other domains, e.g., XML processing. We expect that our approach to taming goto/label will be useful for computation over a mostly hierarchical structure.

[5] Their worst-case analysis is wrong on the size of a graph that they called a *combining DAG*. It can be exponential to the number of nodes in a given CFG, e.g., a sequence of if-then-else statements, whose regular path is $(r_1 \mid r_2) \cdot (r_3 \mid r_4) \cdots$.

# Chapter 8

# Syntax-Directed Construction of Value Graphs

This chapter is self-contained and is an extended version of our unpublished paper [Sat14b].

## 8.1 Introduction

Compiler optimizations are classified roughly into two kinds. One is high-level optimization, which consists of analysis and transformation on abstract syntax trees (ASTs). Although compilers may desugar source languages into simpler ones, ASTs to which compilers apply high-level optimizations contain many language features of source languages, i.e., high-level information. High-level optimizations utilize these features. The other is low-level optimization, which consists of analysis and transformation on control-flow graphs (CFGs) of low-level languages such as three-address code. Low-level languages are much simpler than source languages and designed to be independent both of source languages and machine architectures.

Modern optimizing compilers such as GCC and LLVM[1] are equipped with various low-level optimizations. Especially, optimizations on static single assignment (SSA) form [RWZ88] are de facto standard. Meanwhile, parallelizing compilers [AK01] and recent just-in-time (JIT) compilers [WWS+12, WWW+13, ZLBF14] are equipped with high-level optimizations. Compilers perform high-level optimizations followed by low-level optimizations and do not mix up them. For example, the Truffle JIT framework performs type specialization by rewriting ASTs and then constructs SSA form for underlying Java JIT compilers.

High-level information is useful even in low-level optimizations. For example, high-level control flow on SSA form enables more precise equality detection [AWZ88]. Low-level optimization techniques are also useful in high-level optimizations. Array SSA [KS98], which is an SSA form integrating arrays and the do loops in Fortran is known to be useful for parallelization. Value graphs, which are representations of expressions derived from SSA form, are useful for relatively high-level algebraic transformations [TSTL09]. A mixture of high-level optimizations and low-level optimization techniques is promising.

There are two approaches to this mixture. One is to extract high-level information from low-level representations. For example, we can use control-flow analysis [Sha80, Bak77] for extracting high-level control structures from CFGs. Grosser et al. [GGL12] addressed extracting parallelizable clean loops from low-level SSA form. This approach, however, does not always capture satisfactory high-level information because it would break in low-level representations.

The other is to construct SSA form particularly for high-level optimizations. This is expensive and complicated. SSA form contains $\phi$-functions, which are nonexecutable imaginary operators to represent control flow. At the end of optimizations, we have to eliminate $\phi$-functions adequately in SSA destruction. Moreover in SSA form, $\phi$-functions become an obstacle to code motion. SSA form is not designed

[1] `http://llvm.org/`

for high-level optimizations and the adaptation of SSA form for high-level optimizations is not always straightforward.

To obtain a good mixture of high-level optimizations and low-level optimization techniques, we study the construction of value graphs [AWZ88], which are useful for high-level optimizations. As mentioned above, although value graphs are usually derived from SSA form, we do not have to construct SSA form to obtain value graphs. Because SSA form is not very useful for high-level optimizations, a direct construction of value graphs from ASTs in normal form is advantageous to high-level optimizations.

In this chapter, we present a syntax-directed method for constructing value graphs from ASTs in high-level source languages. We have applied Rosen's syntax-directed approach [Ros77] and function-based approaches [Ros80, Tar81a] in the context of data-flow analysis (DFA) to construction of value graphs. Our method constructs value graphs while placing $\phi$-functions on the basis of high-level control structures, as in the method by Brandis and Mössenböck [BM94] for constructing SSA construction from structured programs. Then, placed $\phi$-functions represent high-level control structures in value graphs. Our method deals with break/continue (or equivalent goto/label) statements in a simple yet efficient manner. Our method can construct value graphs from the while language with break/continue in a single-pass bottom-up manner on a given AST. If we assume arbitrary use of goto/label statements, we face malignant ones that make it difficult to construct value graphs in a syntax-directed manner. We deal with malignant goto/label statements by using reduced ASTs, where benignant parts of ASTs are reduced and malignant parts are exposed. On the basis of reduced ASTs, our method tames malignant goto/label statements with respect to precision or cost. Specifically, our method provides two choices: to perform in a syntax-directed manner incurring imprecision or to perform a little computation in a CFG-based manner. Anyhow, because our method processes much part of a given AST efficiently, our method is more efficient than SSA construction on CFGs. By combining our method for constructing value graphs and syntax-directed methods [Ros77, MFS79, SM14b] for DFA, we can perform value numbering to the whole program as a high-level optimization.

The following are our main contributions:

- We present a syntax-directed method for constructing value graphs from goto-free programs in the while language (Section 8.3). This method constructs value graphs with a composition operator over value graphs in a single-pass bottom-up manner on a given AST while placing $\phi$-functions in value graphs as in the method by Brandis and Mössenböck [BM94] for SSA construction.

- We present an extension for dealing with typical use of goto/label statements such as break/continue ones, in a single-pass bottom-up manner on a given AST (Section 8.4.2). We have extended value graphs for single-entry multiple-exit fragments of ASTs and then applied Rosen's syntax-directed approach in the context of DFA, to construction of value graphs.

- We present two extensions for taming arbitrary, especially malignant use of goto/label statements by using reduced ASTs, where benignant parts of ASTs are reduced (Section 8.4.3). One extension performs in a syntax-directed manner but incurs redundant $\phi$-functions, which degrade the quality of value graphs. The other extension performs a little computation on the basis of the standard algorithm [CFR$^+$91] for placing $\phi$-functions on CFGs. Use of reduced ASTs suppresses both negative effects on value graphs and additional computational costs.

## 8.2 Value Numbering and Value Graphs

### 8.2.1 Value Numbering

We first assume a straight-line (i.e., branchless) language as shown in Figure 8.1 for simplicity and introduce value numbering dealt with in this chapter.

To eliminate redundant expressions, we first have to discover equivalent expressions. Textual representations are inappropriate to the comparison of expression values for two reasons. The first is that the same textual representation may denote different values in different program points. The second is that different textual representations may denote the same value in a program point. A value graph is a

$$
\begin{array}{lcll}
P & ::= & s & \text{(Program)} \\
s & ::= & v \leftarrow e \mid s_1\ s_2 & \text{(Statement)} \\
e & ::= & v \mid c \mid e_1 \circledast e_2 & \text{(Expression)}
\end{array}
$$

Figure 8.1: Syntax of straight-line language, where $v$, $c$, and $\circledast$ denote metavariables over variables, constants, and binary operators, respectively.

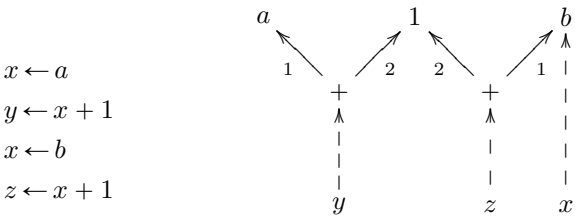



Figure 8.2: Example of value graphs, where the value graph on the right is derived from the branchless program on the left. In the value graph, a labeled solid edge denotes an argument reference of an operator where a label number denotes the position in its argument list, and a break edge denotes a variable reference at the end of the program.

representation of the value structures of expressions. Figure 8.2 shows an example branchless program and the value graph derived from it. The value of $y$ in the example program is $x + 1$ at the second line and that of $z$ is $x + 1$ at the forth line. Both expressions are symbolically the same $x + 1$ but have different values. Meanwhile, each expression corresponds to a node in the value graph. The value graph shows that $y$ and $z$ at the end denote $a + 1$ and $b + 1$ respectively. Value graphs are independent of textual representations and represent expression values directly.

We can detect equivalent expressions on the basis of congruence on graph values. For example, consider the subgraph rooted by the + node to which $y$ refers and that to which $z$ refers, shown in Figure 8.3. Both subgraphs have the identical structure, i.e., are congruent. In other words, every trace from both roots is identical. Therefore, $y$ and $z$ refer to different expressions in the program but refer to the same value. By using congruence on graph values, we thus can construct the equivalence classes of expressions occurred in a given program.

Value numbering [CS70] is to eliminate redundant expressions by using the equivalence on value graphs. Thanks to value graphs, it has an effect similar to the effect of iterations of copy propagation, constant propagation, and common subexpression elimination (CSE) based on textual representations.

In this chapter, we deal with the construction of value graphs. Although the detection of congruence

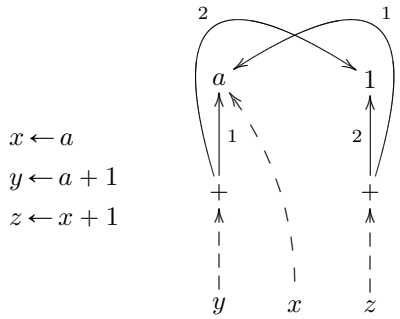



Figure 8.3: Example of congruence of value graphs. The subgraph referred by $y$ and that by $z$ are congruent; thereby the values of $y$ and $z$ are identical.

$$s \quad ::= \quad \ldots \mid s_1\ s_2 \mid \textbf{pass} \mid \textbf{if}\ (e)\ \{s_1\}\ \textbf{else}\ \{s_2\} \mid \textbf{while}\ (e)\ \{s\} \qquad \text{(Statement)}$$

Figure 8.4: Syntax of the while language, which is an extension to the syntax defined in Figure 8.1 and **pass** denotes an empty statement.

is essential for value numbering, we do not deal it. We assume use of existing methods [DST80, AWZ88] for detecting congruence on value graphs. The detection and elimination of redundancy are also necessary for value numbering as an optimization. In fact, these are not essential issues on value numbering. Once the equivalence classes of expressions occurred in a given program are obtained, we can use classic optimization methods based on data-flow analysis (DFA) for detecting and eliminating redundancy [BCS97]. We therefore do not consider the detection and elimination of redundancy.

Constant folding is important for value numbering because it improves the precision of equivalence classes based on congruence. We can implement constant folding straightforwardly in the form of a reduction of value graphs. We assume that constant folding is applied to value graphs after their construction before the detection of congruence.

Note that a mapping $\mathcal{T}$ from each expression to a node in value graphs is necessary to represent equivalence classes of expressions. After detecting congruence on value graphs, we obtain equivalence classes in the nodes of value graphs. $\mathcal{T}$ maps an expression to a node contained in an equivalence class. The comparison of the equivalence classes through $\mathcal{T}$ suffices for checking the equivalence of expressions. We consider $\mathcal{T}$ to be a data structure for implementation and do not include it in value graphs.

### 8.2.2 $\phi$-Function

We here consider the while language, whose syntax is defined in Figure 8.4. Value numbering for branchless program fragments (generally called basic blocks) are directly applicable to program fragments that contain forks of control flow but contain no join (generally called extended basic blocks). This is because the value graph of a program fragment prior to a fork can be shared among the destinations of the fork. The primary issue on value numbering is the join of control flow. To deal with the joins of control flow, we have to represent *control-dependent values*, i.e., values that depend on execution paths. A $\phi$-function [RWZ88] is a pseudo-operator to represent a control-dependent value.

#### Definition

The $\phi$-function, which is also called a pseudo-assignment [SS69], semantically forms the assignment of a control-dependent value to a variable, e.g., $x \leftarrow \phi(v_1, \ldots)$. All $\phi$-functions are conceptually placed at the join points of control flow such as the exit of an if statement and the entry of a while statement. A $\phi$-function $x \leftarrow \phi(v_1, \ldots)$ denotes that possibly different values from its predecessors become confluent via $x$. We therefore call the variable $x$ of $x \leftarrow \phi(v_1, \ldots)$ the *confluent variable*. The arguments of a $\phi$-function are the values of its confluent variable at all its predecessors, where each argument corresponds to each predecessor. For example, consider the following if statement:

$$\textbf{if}\ (e)\ \{x \leftarrow a\}\ \textbf{else}\ \{\textbf{pass}\}.$$

Then, a $\phi$-function $x \leftarrow \phi(a, x_0)$, where $x_0$ denotes the value of $x$ at the entry, is imaginary at the exit above. The essential information that characterizes a $\phi$-function is its program point (i.e., join point) and confluent variable. We therefore equate a $\phi$-function with this pair.

At a $\phi$-function, the different values *may* become confluent. Unnecessary $\phi$-functions are allowed by definition. In placing $\phi$-functions, their minimality becomes an important issue. The standard minimality of $\phi$-functions is based on dominance frontiers [CFR$^+$91]. We call a $\phi$-function a *redundant* one if it breaks the minimality based on dominance frontiers. Note that we distinguish redundant ones from unnecessary ones for $\phi$-functions.

### $\phi$-Function in Value Graphs

In constructing value graphs of programs that contain joins of control flow, we can interpret a $\phi$-function $x \leftarrow \phi(v_1, \ldots)$ as an assignment whose right-hand side is an expression constructed by using $\phi$ as an ordinary operator such as $+$. In detecting congruence on value graphs, we, however, have to interpret $\phi$ in a manner different from ordinary operators. Although we can ignore the confluent variables of $\phi$-functions, we have to take account of the program points of $\phi$-functions.

For example, consider the following sequence of if statements:

$$\textbf{if } (i < n) \ \{x \leftarrow a\} \textbf{ else } \{x \leftarrow b\}$$
$$\textbf{if } (i < m) \ \{y \leftarrow a\} \textbf{ else } \{y \leftarrow b\}.$$

Then, $x \leftarrow \phi(a, b)$ is placed at the exit of the first if statement, and $y \leftarrow \phi(a, b)$ is placed at the exit of the second if statement. If we regarded $\phi$ as an ordinary operator, $x$ and $y$ would refer to the expressions of the same value. This is obviously wrong. Since the values of $n$ and $m$ used in the condition parts of the if statements may be different, the values of $x$ and $y$ cannot be guaranteed to be identical.

To avoid this wrong situation, in detecting congruence on value graphs, we distinguish $\phi$ operators by their program points. To clarify the program point $\pi$ of a $\phi$ operator, we write $\phi_\pi$. In the example above, $x \leftarrow \phi_{\pi_1}(a, b)$ and $y \leftarrow \phi_{\pi_2}(a, b)$ are placed at the end $\pi_1$ of the first line and at that $\pi_2$ of the second, respectively. Since these operator are different, both subgraphs are not congruent. By giving this consideration to $\phi$, we can still construct equivalence classes of expressions on the basis of congruence on value graphs.

### High-Level $\phi$-Function

The cause of this wrong situation that we induce by regarding $\phi$ as an ordinary operator is that $\phi$-functions do not take account of branch conditions of joined control flow. If a $\phi$-function takes the branch condition of joined control flow, its value graph capture the whole control-dependent value from the fork to the join in its structure. The congruence on such value graphs guarantees the equivalence of assigned values as well as branch conditions and thereby guarantees the equivalence of the $\phi$ expressions without their program points.

Since branch conditions and forks of control flow are coupled in control constructs, $\phi$-functions that take branch conditions represent high-level control structures of input languages. To distinguish such $\phi$-functions from normal $\phi$-functions, we call them *high-level $\phi$-functions*[2]. By following the while language defined in Figure 8.4, we introduce the high-level $\phi$-function $\phi_{\text{if}}$ for if statements and $\phi_{\text{while}}$ for while statements, as in Alpern et al.'s work [AWZ88]. We assume that $\phi_{\text{if}}$ and $\phi_{\text{while}}$ contain their source if/while statements as their program points for convenience.

$\phi_{\text{if}}(c, t, e)$ denotes the so-called ternary (conditional) operator (e.g., $c \ ? \ t : e$ in the C language) and conceptually exists at the exit of the source if statement. Since these are pure expressions, it is obvious that its congruence leads to its equivalence. In the example above, the first is $x \leftarrow \phi_{\text{if}}(i < n, a, b)$ and the second is $y \leftarrow \phi_{\text{if}}(i < m, a, b)$. If the value graphs of $n$ and $m$ are congruent, both expressions are identical. Then, the second if statement would be eliminated in value numbering. This elimination is impossible when we use the congruence only on normal $\phi$-functions.

$\phi_{\text{while}}$ has to take account of the circularity of value graphs but is essentially not different from $\phi_{\text{if}}$. $\phi_{\text{while}}(c, s, b)$ conceptually exists at the entry of its source while statement, where $c$, $s$, and $b$ denote the condition part, the initial value, and the recurring body value, respectively. Typically, the subgraphs starting from $c$ and $b$ have circular references to $\phi_{\text{while}}(c, s, b)$.

Note that $\phi_{\text{while}}$ conceptually exists outside of the while loop but it is not a loop-invariant. The value of $\phi_{\text{while}}(c, s, b)$ on the inside of its loop is different from that on the outside. However, if we use only $\phi_{\text{while}}$, we cannot distinguish both expressions by structure of value graphs and thereby both expressions become wrongly equivalent on the basis of congruence. To avoid this wrong situation, we append a sentinel $\phi_{\text{end}}$ to $\phi_{\text{while}}$ at the exit of its while statement, as shown in Figure 8.5.

[2] Alpern et al. [AWZ88] introduced the same notion but did not used the same term. In later studies on SSA, it was called a gating function [BMO90]. We here use the term for emphasizing the high-level part as in the original study.

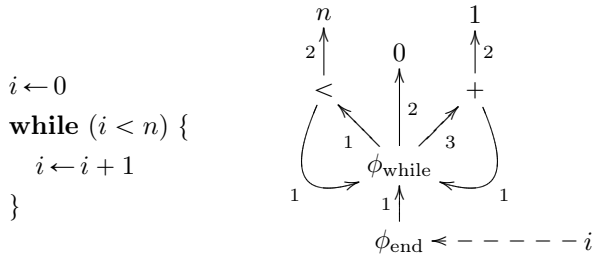



Figure 8.5: Example of circular value graph with $\phi_{\text{while}}$ and $\phi_{\text{end}}$, where $i$ on the right refers to its value at the exit of the while statements on the left.

Another important point in detecting congruence is that we have to distinguish $\phi_{\text{while}}$ of an outer loop in a nested loop from $\phi_{\text{while}}$ of an inner loop. It is sufficient to check containment relation between two while statements in the comparison of $\phi_{\text{while}}$. These considerations do not affect the construction of value graphs. Refer to the discussion on $\phi_{\text{while}}$ and $\phi_{\text{end}}$ in Section 8.5.

### 8.2.3 Definition and Formalization of Value Graphs

#### Definition

Here we formally define the data structure of value graphs in this chapter. A value graph is a triple $(G, V, F)$. $G$ is a directed graph $(N, E)$ whose $N$ is a set of tagged nodes and $E$ is a set of ranked edges. $V$ is a mapping from an alphabet $\Sigma$ to $N$. $F$ is an injective binary relation from $\Sigma$ to $N$.

These denotations in programs are as follows. $\Sigma$ is a set of variable symbols. $G$ is a set of expression values, which is a solid-line part of value graphs shown in Figures 8.2, 8.3, and 8.5. $V$ is a variable binding, which is a break-line part of value graphs. $F$, which we introduce for convenience of algorithm description, captures the occurrences of free variables (i.e., variables of unknown values) in $N$. The rank in $E$ denotes the position in an argument list. The set of tags in $N$ consists of variables, constants operators, and $\phi$-functions. Each node has the only tag but tags are mutable. Multiple nodes can have the same tag. $n^t$ denote a node $n$ tagged by $t$, but we omit $t$ in the case where $t$ is unused. We call a node tagged by a variable a variable node and call a node tagged by a $\phi$-function a $\phi$-node. For every value graph $(G, V, F)$, it is an invariant that $v \mapsto n^v \in F$ for any variable node $n^v \in G$.

For algorithm descriptions, we introduce a special variable \$ and a special map $\omega$. \$ is a variable that refers to the value of an expression and does not appear in a given program. $\omega$ denotes no control flow and pretends a variable binding as $(G, \omega, F)$. $\omega$ is used for dealing with goto statements in Section 8.4.

#### Formalization by Circuits

We also formalize a value graph $(G, V, F)$ as a circuit. We regard $G$ as a black-box circuit where we can observe only input/output points. $\text{img}(V)$, i.e., the image of $V$ corresponds to the output points of $G$, and $\text{img}(F)$ corresponds to the input points of $G$. Wires are connected to the input/output points of $G$ from the outside. Variables symbols in $\text{dom}(V)$ and $\text{dom}(F)$ correspond to the colors of wires to distinguish them. Nodes tagged by constants or variables are terminals in $G$ and nodes tagged by operators or $\phi$-functions are gates in $G$. This formalization (or a view of value graphs) enables us to focus on the concerned part of $G$ on constructing value graphs.

## 8.3 Construction Algorithm to the While Language

We describe our one-pass syntax-directed algorithm for constructing value graphs with high-level $\phi$-functions from programs in the while language. $\mathcal{C}$ denotes a function constructing a value graph from an AST of the while language.

### 8.3.1 Syntax-Directed Formulation

We define $\mathcal{C}$ for each case in this subsection. Let *freshNode*$(t)$ be a primitive function that generates a *fresh* node[3] tagged by $t$. A fresh node is distinct from every other node.

First, from the definition of value graphs, the definitions in the cases of expressions and assignments are immediately obtained.

$$
\begin{aligned}
&\mathcal{C}(v) = ((\{n\}, \varnothing), \{\$ \mapsto n\}, \{v \mapsto n\}),\\
&\quad \text{where } n = \mathit{freshNode}(v),\\
&\mathcal{C}(c) = ((\{n\}, \varnothing), \{\$ \mapsto n\}, \varnothing),\\
&\quad \text{where } n = \mathit{freshNode}(c),\\
&\mathcal{C}(e_1 \circledast e_2) = (G_3, \{\$ \mapsto n_3\}, F_1 \cup F_2),\\
&\quad \text{where } G_3 = G_1 \cup G_2 \cup (\{n_3\}, \{n_3 \mapsto n_1, n_3 \mapsto n_2\}),\\
&\qquad\quad (G_1, \{\$ \mapsto n_1\}, F_1) = \mathcal{C}(e_1),\\
&\qquad\quad (G_2, \{\$ \mapsto n_2\}, F_2) = \mathcal{C}(e_2),\\
&\qquad\quad n_3 = \mathit{freshNode}(\circledast),\\
&\mathcal{C}(v \leftarrow e) = (G, \{v \mapsto n\}, F),\\
&\quad \text{where } (G, \{\$ \mapsto n\}, F) = \mathcal{C}(e).
\end{aligned}
$$

For simplicity, the algorithm description above takes no account of the order of the operands of $\circledast$. It is immediate to take account of operand order. Similarly, in the remainder of this chapter, we use algorithm descriptions taking no account of operand order.

The case of statement sequencing $s_1\ s_2$ is defined as

$$\mathcal{C}(s_1\ s_2) = \mathcal{C}(s_1) \otimes \mathcal{C}(s_2),$$

where $\otimes$ is a serial composition operator over value graphs. The meaning of $\otimes$ is easy to understand as the construction of series circuits on the basis of the formalization by circuits. Specifically, it is to construct correct wire connections between the output points of $\mathcal{C}(s_1)$ and the input points of $\mathcal{C}(s_2)$. The outgoing wire in a color of $\mathcal{C}(s_1)$ connects to all the incoming wire in the same color $\mathcal{C}(s_2)$. Then, we remove terminals in $\mathcal{C}(s_2)$ connected to $\mathcal{C}(s_1)$ and short-circuit incoming wires in $\mathcal{C}(s_2)$. We can formulate $\otimes$ over the data structures of value graphs as follows:

$$
\begin{aligned}
&((N_1, E_1), V_1, F_1) \otimes ((N_2, E_2), V_2, F_2) = (G_3, V_3, F_3),\\
&\quad \text{where } G_3 = (N_1 \cup \mathit{update}_\sigma(N_2), E_1 \cup E_2' \cup E_\Delta),\\
&\qquad V_3 = \{v \mapsto n \in V_1 \mid v \notin \mathrm{dom}(V_2)\} \cup V_2,\\
&\qquad F_3 = F_1 \cup \{v \mapsto n \in F_2 \mid v \notin S\} \cup \{v' \mapsto n_2 \mid v \mapsto n_2 \in F_{\mathrm{alias}},\ n_1^{v'} = V_1(v)\},\\
&\qquad F_{\mathrm{alias}} = \{v \mapsto n_2 \in F_2 \mid v \in S,\ n_1^{v'} = V_1(v)\},\\
&\qquad E_\Delta = \{n' \mapsto V_1(v) \mid n' \mapsto n \in E_2,\ v \mapsto n \in F_2 - F_{\mathrm{alias}},\ v \in S\},\\
&\qquad E_2' = E_2 - \{n' \mapsto n \in E_2 \mid v \mapsto n \in F_2 - F_{\mathrm{alias}},\ v \in S\},\\
&\qquad \sigma = \{v \mapsto v' \mid v \mapsto n_2 \in F_{\mathrm{alias}},\ n_1^{v'} = V_1(v)\},\\
&\qquad S = \mathrm{dom}(V_1) \cap \mathrm{dom}(F_2).
\end{aligned}
$$

$\mathit{update}_\sigma(N)$ is a primitive operation to update the tags of variable nodes in a set $N$ according a mapping $\sigma$ over $\Sigma$. For example, $\mathit{update}_{\{v \mapsto v'\}}(\{n_1^+, n_2^v\}) = \{n_1^+, n_2^{v'}\}$. Note that the identity of $n_2$ here is unchanged and the edges of $n_2$ remain connected after updating. $\mathit{update}_\sigma(N_2)$ corresponds to copy propagation on the nodes of a resultant value graph. $F_{\mathrm{alias}}$ is the subset of $F_2$ sensitive to this copy propagation. The third term in the RHS of the definition of $F_3$ makes the invariant $v \mapsto n^v \in F_3$ hold.

[3]Node tagged by constants do not have to be fresh regarding the same constant.

When we regard a circuit as a function, the serial composition $\otimes$ corresponds to function composition. $\otimes$ is therefore associative and its identity is $(\varnothing, \varnothing, \varnothing)$. Since this identity corresponds to the identity function, this meaning is to do nothing. The case of empty statements is therefore defined as

$$\mathcal{C}(\textbf{pass}) = (\varnothing, \varnothing, \varnothing).$$

To define the cases of if statements and while statements, we have to place high-level $\phi$-functions conceptually. This is straightforward by using the single-entry single-exit control flow of these high-level control structures [BM94].

Specifically, the single-entry single-exit control flow in if statements is observed as follows. To reach the then/else part of an if statement, it is necessary to pass through the entry of the if statement; it means single-entry. To reach the exit of an if statement, it is necessary to pass through the exit of the then/else part; it means single-exit. These lead to two observations. $\phi$-functions at the exit of an if statement are unnecessary for variables defined neither in the then part nor the else part. If a variable $v$ is defined either in the then part or the else part, the value of $v$ at the entry reaches the exit and may become confluent via $v$. Without redundant (high-level) $\phi$-functions, we can therefore define the case of if statements as follows:

$$\begin{aligned}
&\mathcal{C}(\textbf{if}\ (e)\ \{s_1\}\ \textbf{else}\ \{s_2\}) = (G, V, F),\\
&\quad \text{where } G = G_c \cup G_t \cup G_e \cup (\mathrm{img}(V) \cup \mathrm{img}(F_\Delta), E_\Delta),\\
&\qquad V = \{v \mapsto \mathit{freshNode}(\phi_{\mathrm{if}}) \mid v \in \Sigma_\phi\},\\
&\qquad F = F_c \cup F_t \cup F_e \cup F_\Delta,\\
&\qquad F_\Delta = \{v \mapsto \mathit{freshNode}(v) \mid v \in \Sigma_\phi - (\mathrm{dom}(V_t) \cap \mathrm{dom}(V_e))\},\\
&\qquad E_\Delta = \{V(v) \mapsto n_c \mid v \in \Sigma_\phi\} \cup \{V(v) \mapsto n \mid v \mapsto n \in F_\Delta\},\\
&\qquad\qquad \cup \{V(v) \mapsto n \mid v \mapsto n \in V_t\} \cup \{V(v) \mapsto n \mid v \mapsto n \in V_e\},\\
&\qquad \Sigma_\phi = \mathrm{dom}(V_t) \cup \mathrm{dom}(V_e),\\
&\qquad (G_c, \{\$ \mapsto n_c\}, F_c) = \mathcal{C}(e),\\
&\qquad (G_t, V_t, F_t) = \mathcal{C}(s_1),\\
&\qquad (G_e, V_e, F_e) = \mathcal{C}(s_2).
\end{aligned}$$

Here, $\mathit{freshNode}(v)$ denotes the value of $v$ at the entry of a given if statement.

The generation of $\phi_{\mathrm{while}}$ is almost the same as that of $\phi_{\mathrm{while}}$ as an if statement whose else part is an empty statement. The main difference is that the join point of $\phi_{\mathrm{while}}$ is the entry of its while statement. We therefore have to consider the sequencing of $\phi_{\mathrm{while}}$ derived from a while statement followed by the while statement. We apply serial composition $\otimes$ to the sequencing, and thereby we introduce circularity into a resultant value graph. Finally, we append a $\phi_{\mathrm{end}}$ node to $\phi_{\mathrm{while}}$ node. The case of while statements is defined as follows:

$$\begin{aligned}
&\mathcal{C}(\textbf{while}\ (e)\ \{s\}) = (G \cup (\mathrm{img}(V_{\mathrm{end}}), E_{\mathrm{end}}), V_{\mathrm{end}}, F),\\
&\quad \text{where } E_{\mathrm{end}} = \{V_{\mathrm{end}}(v) \mapsto V(v) \mid v \in \mathrm{dom}(V)\},\\
&\qquad V_{\mathrm{end}} = \{v \mapsto \mathit{freshNode}(\phi_{\mathrm{end}}) \mid v \in \mathrm{dom}(V)\},\\
&\qquad (G, V, F) = ((\mathrm{img}(V_\Delta) \cup \mathrm{img}(F_\Delta), E_\Delta), V_\Delta, F_\Delta) \otimes (G_c \cup G_b, \varnothing, F_c \cup F_b),\\
&\qquad V_\Delta = \{v \mapsto \mathit{freshNode}(\phi_{\mathrm{while}}) \mid v \in \mathrm{dom}(V_b)\},\\
&\qquad F_\Delta = \{v \mapsto \mathit{freshNode}(v) \mid v \in \mathrm{dom}(V_b)\},\\
&\qquad E_\Delta = \{n' \mapsto n_c,\ n' \mapsto F_\Delta(v),\ n' \mapsto n \mid v \mapsto n \in V_b,\ n' = V_\Delta(v)\},\\
&\qquad (G_c, \{\$ \mapsto n_c\}, F_c) = \mathcal{C}(e),\\
&\qquad (G_b, V_b, F_b) = \mathcal{C}(s).
\end{aligned}$$

Here, $\mathit{freshNode}(v)$ denotes the value of $v$ at the entry of a given while statement.

### 8.3.2 Implementation Issues

#### Use of Union-Find Trees

Although $\mathcal{C}$ achieves the construction of value graphs from ASTs in a syntax-directed manner, a naive implementation of $\mathcal{C}$ can cause a problem on efficiency. $update_\sigma(N_2)$ in the definition of $\otimes$ corresponds to a naive copy propagation. The time complexity of this naive copy propagation depends on the order of processing assignments. For example, consider the following sequence of assignments:

$$\begin{aligned} x_1 &\leftarrow a \\ x_2 &\leftarrow x_1 \\ &\vdots \\ x_m &\leftarrow x_{m-1}. \end{aligned}$$

A naive copy propagation is to rewrite a RHS variable with a LHS variable in nearest prior assignments. It costs $\mathrm{O}(m)$ time if we perform it from $x_2 \leftarrow x_1$, but it costs $\mathrm{O}(m^2)$ time if we perform it from $x_m \leftarrow x_{m-1}$. $\mathcal{C}$, which performs in the simplest syntax-directed manner, does not guarantee that its processing order is the program order. If we complicated $\mathcal{C}$, we could guarantee it but shall spoil the simplicity of its syntax-directed manner. This inefficiency of $\mathcal{C}$ is, however, unacceptable.

The cause of this inefficiency is to perform copy propagation eagerly. If we perform it lazily, we can avoid the worst case. We can consider variable rewriting as construction of equivalence classes and therefore we can implement this lazy copy propagation by using union-find trees[4].

Specifically, we construct an equivalence class of the nodes in tagged by the same variable by using a union-find tree. As a result, since the invariant $v \mapsto n^v \in F$ holds, $F$ becomes an injective function. We require two simple modifications on $\mathcal{C}$. First, when we construct $F_1 \cup F_2$, letting $v \in \mathrm{dom}(F_1) \cup \mathrm{dom}(F_2)$, we unify two nodes $F_1(v)$ and $F_2(v)$. Second, when we refer to variable nodes, we find their representatives. To update the tags of only representatives suffices for copy propagation. We therefore use the only field of each variable node both for a tag and a link to representative.

If we use union-find trees equipped with path compression and union by rank, the copy propagation for $m$ assignments costs $\mathrm{O}(m\alpha(m,m))$, where $\alpha(m,m)$ denotes the inverse Ackermann function. From a realistic size of $m$, we can assume $\alpha(m,m)$ to be a small constant. $\mathcal{C}$ thus becomes almost as efficient as ordinary program-order algorithms.

Use of union-find trees also simplifies the implementation of $\mathcal{C}$. In particular, it simplifies the construction of $E_\Delta$ in the definition of $\otimes$. A naive way of implementing it is that all variable nodes mange reverse edges. This requires mutation of the fields of operator nodes and more fields in a variable node. Meanwhile, if we change the representative of a variable node to operator nodes in applying $\otimes$, we achieve the equivalent result of $E_\Delta$ in later find operations.

#### Reference from ASTs to Value Graphs

As mentioned in Section 8.2.1, for value numbering, we require a mapping $\mathcal{T}$ from expressions in ASTs to nodes in value graphs. Since $\mathcal{C}$ constructs a distinct node for each expression, $\mathcal{T}$ becomes an injective function. We can share $\mathcal{T}$ globally and implicitly in $\mathcal{C}$. A simple and efficient way of constructing $\mathcal{T}$ is to register a fresh node corresponding to a given expression with globally visible $\mathcal{T}$, in the case of expressions regarding $\mathcal{C}$. We here assume $\mathcal{T}$ to be a separate table because it is easy to construct and dispose of. We also can make each node in ASTs hold a reference to value graphs. An actual representation of $\mathcal{T}$ depends on implementation.

That $\mathcal{T}$ holds references to nodes of value graphs simplifies the implementation of a value graph $(G, F, V)$. If we construct $G$ as a pointer structure, we can reach nodes of $G$ through $\mathcal{T}$, and thereby we do not have to construct $G$ explicitly as a set. An important observation is that optimizations concern the part of $G$ reachable from $\mathcal{T}$. In particular, $\phi$-functions for dead variables are unnecessary. Actually, existing algorithms [BCHS98, CCF91] for placing $\phi$-functions prune unnecessary $\phi$-functions on the basis

[4]The original paper [GF64] of union-find trees also aimed at managing equivalent variables in compilers.

$$s \quad ::= \quad \ldots \mid l\text{:} \mid \textbf{goto}\ l \qquad \text{(Statement)}$$

Figure 8.6: Syntax of the while language with goto/label statements, which is an extension to the syntax defined in Figure 8.4, where $l$ denotes a metavariable over labels. Label statements that are not targeted by any goto statement are regarded as empty statements.

of live variable analysis. We, however, can prune $\phi$-functions easily on the basis of the reachability from $\mathcal{T}$. When we trace links through $\mathcal{T}$ from all essential expressions (e.g., returned expressions), unreachable $\phi$-functions from $\mathcal{T}$ are simply unnecessary. If implementation languages have garbage collection (GC), unnecessary $\phi$-functions can be implicitly removed by using weak references.

## 8.4 Taming Goto/Label Statements

Goto statements are not considered to be favorable, are not adopted in many languages because structured programming is now widespread. An appropriate use of goto statements is useful even in structured programming [Knu74]. Although use of goto statements decreases, they actually have been used in source code yet [SW12]. Moreover, it is not rare that compiler frontends introduce goto/label statements. Goto statements are still not negligible for compilers. In this section, we describe the difficulty of goto/label statements and deal with the construction of value graphs from programs in the while language with goto/label defined in Figure 8.6.

### 8.4.1 Difficulty of Goto/Label Statements

#### Analogy and Difference between DFA and Value-Graph Construction

In Section 8.3, we define serial composition operator $\otimes$ by formalizing a value graph as a circuit. As mentioned earlier, a circuit is a kind of functions. To construct functions like value graphs from programs is called a function-based approach in the context of DFA, and known to be synergistic with syntax-directed approaches [Ros77, Ros80, SM14b]. We have already developed a syntax-directed method for taming goto/label statements in DFA [SM14b]. It is inadequate for the construction of value graphs despite the analogy between both formalizations. A matter is the $\phi$-function placement.

In monotone DFA, we can define an addition $\oplus$ over data-flow functions, where function composition $\circ$ is left-distributive[5] over $\oplus$, i.e., $f_3 \circ (f_2 \oplus f_1) = (f_3 \circ f_2) \oplus (f_3 \circ f_1)$. This means that the data-flow function of the following program

$$\textbf{if}\ (e)\ \{x \leftarrow a\}\ \textbf{else}\ \{x \leftarrow b\}\ y \leftarrow x + 1$$

is the same as that of the following program

$$\textbf{if}\ (e)\ \{x \leftarrow a \quad y \leftarrow x + 1\}\ \textbf{else}\ \{x \leftarrow b \quad y \leftarrow x + 1\}.$$

However, the value graphs of both programs are not the same with respect to $\phi$-functions. As seen from this difference, it is essentially difficult to define an addition over value graphs as in the algebra of DFA. Our method [SM14b] relies heavily on the left-distributive law and therefore inadequate for the construction of value graphs.

#### Troubles of the $\phi$-Functions Placement for Goto/Label Statements

This trait on $\phi$-functions make it very difficult to construct value graphs without settling the $\phi$-function placement for label statements. For example, consider the following program fragment:

$$\textbf{goto}\ l_1\ l_2\text{:}.$$

[5] Strictly, the distribution of $\circ$ will improve the precision of DFA and the left-distributive law should be an inequality.

A simplest syntax-directed manner like $\mathcal{C}$ is to define a useful value graph of the fragment above without any additional information. This is, of course, almost impossible because we do not have any knowledge of the $\phi$-function placement. Then, we assume the information of this context to be given; e.g., what variables are defined before the fragment and used after the fragment. However, it is still very difficult to place reasonable $\phi$-functions because the control flow outgoing from **goto** $l_1$ and incoming to $l_2$: is independent of this context. For example, consider that a given program contains the only **goto** $l_2$. Any $\phi$-function at $l_2$: becomes redundant but the variable binding incoming to $l_2$: may necessitate $\phi$-functions after the fragment. It is unclear what information is adequate to place $\phi$-functions.

The essence of syntax-directed manners is to define calculation to a subtree of a given AST. The $\phi$-function placement for goto/label statements is inherently difficult to perform in a syntax-directed manner. A reasonable syntax-directed method for constructing value graphs from the while language with goto/label is therefore nontrivial.

### 8.4.2 Syntax-Directed Approach to Single-Entry Multiple-Exit ASTs

Classifying control flow on syntactic structures is known to be useful for taming goto/label statements [MFS79]. We consider three classifications: *single-entry single-exit* (SESE) ASTs, *single-entry multiple-exit* (SEME) ASTs, and *multiple-entry* ASTs. An AST is SESE if any control transfer into/from the AST must pass through its AST. The while language always produces SESE ASTs. An AST is SEME if any control transfer into the AST must pass through its root. An AST is multiple-entry if there is a control transfer into the AST without passing through its root. It is inherently difficult to deal with ME ASTs. We describe how to deal with multiple-entry ASTs later. We here deal with single-entry multiple-exit ASTs.

Typical SEME ASTs are produced from the while language extended with break/continue statements such as the ones in C. In the while language with goto/label, SEME ASTs formally mean that any **goto** $l$ is contained in $s$ either of $l$: $s$ or $s$ $l$:. The case of $l$: $s$ and that of $s$ $l$: correspond to continue statements and break statements, respectively. Many of useful usages of goto statements in structured programming claimed by Knuth [Knu74] form this kind of usage. Rosen's method [Ros77] for DFA deals with SEME ASTs in a clear syntax-directed manner. In the rest of this subsection, we extend $\mathcal{C}$ to $\mathcal{C}'$ for SEME ASTs on the basis of Rosen's syntax-directed approach.

From the perspective of syntax-directed computations, Rosen's method sets aside data flow outgoing from goto statements in descendants and merges the reserved data flow with the data flow ignoring goto statements at the current node. This method is applicable to placing $\phi$-functions because $\phi$-functions are unnecessary for a variable $v$ if $v$ is not defined in the reachable part of the current AST. The main difference is that we have to distinguish all goto statements because the value of a variable incoming from each goto statement becomes a distinct argument of a $\phi$-function.

The information that we have to set aside is a variable binding $V$ in a value graph $(G, V, F)$. We extend $(G, V, F)$ by adding set-aside variable bindings. An extended value graph is a quadruple $(G, V, \mathbf{V}, F)$, where $\mathbf{V}$ is a mapping from a goto statement to the variable binding at the goto statement and $(G, V, F)$ is an original value graph, which ignores the effect of goto statements. While an original value graph is formalized as a circuit having a single bundle of outgoing wires, an extended value graph is formalized a circuit having multiple bundles of outgoing wires, where each bundle contains at most one wire in each color. We internally confuse $(G, V, \varnothing, F)$ with $(G, V, F)$. $l^\pi$ denotes a goto statement **goto** $l$ at the program point $\pi$.

We define $\mathcal{C}'$, which takes an AST in the while language with goto/label yields an extended value graph, as follows. We can define the case of goto statements as

$$\mathcal{C}'(\textbf{goto } l) = (\varnothing, \omega, \{l^\pi \mapsto \varnothing\}, \varnothing),$$

where $\pi$ denotes the program point of a given goto statement. As mentioned in Section 8.2.3, $\omega$ denotes no control flow. If $\mathcal{C}'(s) = (G, \omega, \mathbf{V}, F)$, there is no reachable path from the entry of $s$ to the exit of $s$.

For statement sequencing $s_1$ $s_2$, where neither $s_1$ nor $s_2$ is a label statement, we have three observations. The first is that, if there is no control flow outgoing form $s_1$, $s_2$ has no effect. The second is that, if there is no control flow outgoing form $s_2$, there is no control flow outgoing form $s_1$ $s_2$ except for

goto-derived control flow. The third is that the variable binding at the exit of $s_1$ has also effect on the variable bindings at the goto statements in $s_2$. From these observations, we can define serial composition operator $\otimes'$ over extended value graphs as follows:

$$(G_1, \omega, \mathbf{V}_1, F_1) \otimes' (G_2, V_2, \mathbf{V}_2, F_2) = (G_1, \omega, \mathbf{V}_1, F_1),$$
$$(G_1, V_1, \mathbf{V}_1, F_1) \otimes' (G_2, V_2, \mathbf{V}_2, F_2) = \begin{cases} (G_3, \omega, \mathbf{V}_3, F_3) & (V_2 = \omega), \\ (G_3, V_3, \mathbf{V}_3, F_3) & (\text{otherwise}), \end{cases}$$
$$\text{where } (G_3, V_3, F_3) = \begin{cases} (G_1, V_1, F_1) \otimes (G_2, \varnothing, F_2) & (V_2 = \omega), \\ (G_1, V_1, F_1) \otimes (G_2, V_2, F_2) & (\text{otherwise}), \end{cases}$$
$$\mathbf{V}_3 = \mathbf{V}_1 \cup \{l^\pi \mapsto \mathit{append}(V_1, V) \mid l^\pi \mapsto V \in \mathbf{V}_2\},$$
$$\mathit{append}(V_1, V) = V \cup \{v \mapsto n \in V_1 \mid v \notin \mathrm{dom}(V)\}.$$

$\otimes'$ is associative and its identity is $(\varnothing, \varnothing, \varnothing, \varnothing)$, which is also the identity of $\otimes$. Since neither $s_1$ nor $s_2$ is a label statement, we do no have to place $\phi$-functions for $s_1\ s_2$. We therefore can define the case of $s_1\ s_2$ simply with $\otimes'$ as

$$\mathcal{C}'(s_1\ s_2) = \mathcal{C}'(s_1) \otimes' \mathcal{C}'(s_2).$$

The $\phi$-function placements for $s\ l$: and $l$: $s$ are analogous to that for if statements and while statements, respectively. We place normal $\phi$-functions at label statements because goto statements have no branch condition. Let $\phi_l$ be a normal $\phi$-function at the label statement $l$:.

Specifically, we can consider $s\ l$: as a multiway nondeterministic branching statement where the control flow outgoing from $s$ and the control flow outgoing from all **goto** $l$ statements are joined at $l$:. We therefore can define the case of $s\ l$: as follows:

$$\mathcal{C}'(s\ l{:}) = (G', V', \mathbf{V} - \mathbf{V}_l, F \cup F_\Delta),$$
$$\text{where } G' = G \cup (\mathrm{img}(V_\Delta) \cup \mathrm{img}(F_\Delta), E_\Delta),$$
$$V' = \{v \mapsto V_0(v) \mid v \in \mathrm{dom}(V_0) - \Sigma_\phi\} \cup V_\Delta,$$
$$V_\Delta = \{v \mapsto \mathit{freshNode}(\phi_l) \mid v \in \Sigma_\phi\},$$
$$F_\Delta = \left\{ v \mapsto \mathit{freshNode}(v) \;\middle|\; v \in \Sigma_\phi - \bigcap_{i=0}^{k} \mathrm{dom}(V_i) \right\},$$
$$E_\Delta = \left\{ V_\Delta(v) \mapsto F_\Delta(v) \mid v \in \Sigma_\phi \right\} \cup \bigcup_{i=0}^{k} \left\{ n \mapsto n' \mid v \mapsto n \in V_\Delta,\ v \mapsto n' \in V_i \right\},$$
$$\Sigma_\phi = \bigcup_{i=0}^{k} \mathrm{dom}(V_i) - \left\{ v \in \bigcap_{i=0}^{k} \mathrm{dom}(V_i) \;\middle|\; V_0(v) = V_1(v) = \cdots = V_k(v) \right\},$$
$$\bigcup_{i=1}^{k} \{l^{\pi_i} \mapsto V_i\} = \mathbf{V}_l,$$
$$\mathbf{V}_l = \{l'^\pi \mapsto V \in \mathbf{V} \mid l' = l\},$$
$$(G, V_0, \mathbf{V}, F) = \mathcal{C}'(s).$$

The definition above adds a multiway extension to the case of if statements in $\mathcal{C}$ except for the second term of the RHS in the definition of $\Sigma_\phi$. The term has effect that excludes the $\phi$-functions for variables that are defined with the identical value. For example, consider the following if statement:

$$x \leftarrow a$$
$$\textbf{if } (i < m)\ \{\textbf{goto } l\}\ \textbf{else}\ \{\textbf{pass}\}$$
$$l{:}.$$

The different values of $v$ are never confluent at $l$:. If we placed the $\phi$-function for $v$ at $l$:, it would be redundant. The additional term takes account of this redundant placement.

We can consider $s\ l$: as a multiway nondeterministic do-while statements where $l$: is the loop header and each **goto** $l$ constitutes a one-way loop. We therefore can define the case of $s\ l$: as follows:

$$
\begin{aligned}
&\mathcal{C}'(l\colon s) = (G_\Delta, V_\Delta, \varnothing, F_\Delta) \otimes' (G, V', \mathbf{V} - \mathbf{V}_l, F),\\
&\quad \text{where } G_\Delta = (\mathrm{img}(V_\Delta) \cup \mathrm{img}(F_\Delta), E_\Delta),\\
&\qquad V' = \{v \mapsto n \in V_0 \mid v \notin \Sigma_\phi\},\\
&\qquad V_\Delta = \{v \mapsto \mathit{freshNode}(\phi_l) \mid v \in \Sigma_\phi\},\\
&\qquad F_\Delta = \{v \mapsto \mathit{freshNode}(v) \mid v \in \Sigma_\phi\},\\
&\qquad E_\Delta = \{V_\Delta(v) \mapsto F_\Delta(v) \mid v \in \Sigma_\phi\} \cup \bigcup_{i=1}^{k} \left\{n \mapsto n' \;\middle|\; v \mapsto n \in V_\Delta,\ v \mapsto n' \in V_i\right\}\\
&\qquad \Sigma_\phi = \bigcup_{i=1}^{k} \mathrm{dom}(V_i),\\
&\qquad \bigcup_{i=1}^{k} \{l^{\pi_i} \mapsto V_i\} = \mathbf{V}_l,\\
&\qquad \mathbf{V}_l = \{l'^{\pi} \mapsto V \in \mathbf{V} \mid l' = l\},\\
&\qquad (G, V_0, \mathbf{V}, F) = \mathcal{C}'(s).
\end{aligned}
$$

The case of $l$: $s$ does not necessitate a special attention to the values of variables defined in $s$ because the values of defined variables become confluent with unknown values at the entry of $l$: $s$.

We consider the cases of if statements and while statements. The primary concern is $\omega$ because set-aside variable bindings have no effect on the confluence of values on if statements and while statements. Specifically, if there is no control flow from the then part of an if statement or the else part, we do not have to place $\phi$-functions at the exit of the if statement. if there is no control flow from the loop body of a while statement, we do not have to place $\phi$-functions at the exit of the while statement. Anyhow, the term $\mathbf{V}$ of a result $(G, V, \mathbf{V}, F)$ is constructed by unifying every term $\mathbf{V}$ of the results of subtrees. We therefore can define the case of if statements and while statements as follows:

$$
\begin{aligned}
&\mathcal{C}'(\textbf{if}\ (e)\ \{s_1\}\ \textbf{else}\ \{s_2\}) = (G, V, \mathbf{V}_t \cup \mathbf{V}_e, F),\\
&\quad \text{where } G = \begin{cases} G_c \cup G_t \cup G_e & (V_t = \omega \vee V_e = \omega),\\ \text{identical to } \mathcal{C} & (\text{otherwise}),\end{cases}\\
&\qquad V = \begin{cases} \omega & (V_t = V_e = \omega),\\ V_e & (V_t = \omega \wedge V_e \neq \omega),\\ V_t & (V_t \neq \omega \wedge V_e = \omega),\\ \text{identical to } \mathcal{C} & (\text{otherwise}),\end{cases}\\
&\qquad F = \begin{cases} F_c \cup F_t \cup F_e & (V_t = \omega \vee V_e = \omega),\\ \text{identical to } \mathcal{C} & (\text{otherwise}),\end{cases}\\
&\qquad (G_c, \{\$ \mapsto n_c\}, F_c) = \mathcal{C}(e),\\
&\qquad (G_t, V_t, \mathbf{V}_t, F_t) = \mathcal{C}'(s_1),\\
&\qquad (G_e, V_e, \mathbf{V}_e, F_e) = \mathcal{C}'(s_2),\\
&\mathcal{C}(\textbf{while}\ (e)\ \{s\}) = \begin{cases} (G_c \cup G_b, \varnothing, \mathbf{V}_b, F_c \cup F_b) & ((G_b, \omega, \mathbf{V}_b, F_b) = \mathcal{C}'(s)),\\ (G, V, \mathbf{V}_b, F) & ((G_b, V_b, \mathbf{V}_b, F_b) = \mathcal{C}'(s)),\end{cases}\\
&\quad \text{where } (G_c, \{\$ \mapsto n_c\}, F_c) = \mathcal{C}(e),\\
&\qquad G,\ V, \text{ and } F \text{ are identical to ones in } \mathcal{C}.
\end{aligned}
$$

The ASTs of expressions, assignments, and empty statements never contain any goto statement. These cases in $\mathcal{C}'$ are identical to $\mathcal{C}$.

$$
\begin{array}{rcll}
P & ::= & s & \text{(Program)} \\
s & ::= & \mathcal{G}' \mid s_1\ s_2 \mid \textbf{if}\ (\mathcal{G})\ \{s_1\}\ \textbf{else}\ \{s_2\} \mid \textbf{while}\ (\mathcal{G})\ \{s\} \mid l: & \text{(Statement)}
\end{array}
$$

Figure 8.7: Syntax of reduced ASTs derived from the while language with goto/label, where $\mathcal{G}$ denotes a metavariable over value graphs and $\mathcal{G}'$ denotes a metavariable over extended value graphs.

### 8.4.3 Tolerating Multiple-Entry ASTs by Reduction

#### Reduced ASTs

The source of the difficulty is *malignant* goto/label statements that cause multiple-entry ASTs. Although compilers can introduce goto/label statements, transformations seldom introduce malignant ones[6]. Especially, high-level transformations such as inlining hardly do it. Since structured programming is widespread, we can assume malignant goto/label statements in a given AST to be rare. We call this assumption *sparse malignancy*.

To utilize sparse malignancy, we introduce a *reduced AST*, which is the result of applying $\mathcal{C}'$ to as large portion of a given AST in the while language with goto/label as possible. Figure 8.7 shows the syntax of reduced ASTs. We can construct reduced ASTs simply by applying $\mathcal{C}'$ to ASTs and by checking whether label statements can reduce to value graphs.

The number of (extended) value graphs in a reduced AST, which we call it the size of the reduced AST, is in proportion to the number malignant label statements in its original AST. Letting $k$ be the number of malignant label statements in a given AST and $d$ be the maximum depth of them, The number of (extended) value graphs in its reduced AST is $\mathrm{O}(kd)$. Owing to the associativity of $\otimes'$, we can flatten statement sequencing, and thereby $d$ is the number of if/while nesting. From the structure of realistic programs, we can assume $d$ to be a constant. From the assumption of sparse malignancy, we can assume $k$ to be small. The sizes of reduced ASTs are therefore small.

The remainder of this subsection, we use (reasonably small) reduced ASTs for placing $\phi$-functions.

#### Approximate Placement of $\phi$-Functions

Although we explain the difficulty of the $\phi$-function placement in Section 8.4.1, strictly, it is difficult to avoid placing redundant $\phi$-functions. As mentioned in Section 8.2.2, placing redundant $\phi$-functions is safe. However, they will be obstacle of detecting congruence on value graphs and thereby degrade the precision of equivalence based on congruence. We present a simple method for approximate placement of $\phi$-functions containing reasonably few redundant ones.

The standard approximate approach is a brute-force placement on CFGs. It places $\phi$-functions for all variables at the entry of every basic block. Our approximate approach to $\phi$-function placement on reduced ASTs is more precise and efficient.

First, the reduced AST of a given program is much smaller that the CFG. because an SEME AST that reduces to an extended value graph is more general than a basic block and contain a larger portion of a given program. The reduced AST displays their join points in its control constructs. Even a brute-force approach on reduced ASTs is therefore more efficient and precise.

Next, we consider pruning $\phi$-functions on the basis of live variables because $\phi$-functions for dead variables are unnecessary. Although we can calculate live variables by using our syntax-directed method [SM14b] for DFA, we can obtain approximate live variables in a simpler manner. We construct merely the union of free variables $\mathrm{dom}(F)$ for all (extended) value graphs. We can use it for a safe approximation of live variables. This approximation is analogous to the construction of live variable in semi-pruned SSA [BCHS98]. This approximation is known to provide a reasonable precision with a cheap computation. While the approximation in semi-pruned SSA is based on CFGs, our approximation is based on reduced ASTs. Since reduced ASTs are always smaller than CFGs of basic blocks, our approximation is more

[6] For example, the elimination of jumps by code replication [MW92, MW95] can change natural loops to unstructured loops. These are applied to programs in the low-level phase.

efficient. While the approximation in semi-pruned SSA calculates live variables precisely in each basic block, our approximation calculates them precisely in the fragment reduced to each extended value graph. Since this fragment covers a larger portion than a basic block, our approximation is more precise.

We obtain a set $\Sigma_\phi$ of approximate live variables by using the method above. Then, we place $\phi$-functions for all variables in $\Sigma_\phi$ at every label statement. If we calculate live variables through DFA, we obtain a distinct set of live variables for each label statement, and thereby place $\phi$-functions for the variables in each distinct set.

**Construction of Value Graphs** Here we define $\mathcal{C}_\mathrm{A}$, where denotes a function that construct a value graph from a reduced AST. We assume $\Sigma_\phi$ to be obtained. Let $\Phi_l(v)$ be a $\phi$-node placed for a variable $v$ at a label statement $l$:. The case of $\mathcal{G}'$ is defined as follows:

$$\begin{aligned}
&\mathcal{C}_\mathrm{A}(G, V, \mathbf{V}, F) = (G', V', F'),\\
&\quad \text{where } G' = G \cup (\mathrm{dom}(E_\Delta) \cup \mathrm{img}(E_\Delta), E_\Delta),\\
&\qquad V' = \{v \mapsto n \in V \mid v \in \Sigma_\phi\},\\
&\qquad F' = F \cup \{v \mapsto \mathit{freshNode}(v) \mid v \in \Sigma_\phi,\ v \notin \mathrm{dom}(F)\},\\
&\qquad E_\Delta = \{\Phi_l(v) \mapsto F'(v) \mid v \in \Sigma_\phi,\ l^\pi \in \mathrm{dom}(\mathbf{V})\},\\
&\qquad\quad \cup \{\Phi_l(v) \mapsto n \mid l^\pi \mapsto V_i \in \mathbf{V},\ v \mapsto n \in V_i\}.
\end{aligned}$$

Here, $\mathit{freshNode}(v)$ denotes the value of $v$ at the entry of the fragment reduced to a given extended value graph. The case of label statements is defined as follows:

$$\begin{aligned}
&\mathcal{C}_\mathrm{A}(l{:}) = ((\mathrm{img}(V) \cup \mathrm{img}(F), E), V, F),\\
&\quad \text{where } V = \{v \mapsto \Phi_l(v) \mid v \in \Sigma_\phi\},\\
&\qquad F = \{v \mapsto \mathit{freshNode}(v) \mid v \in \Sigma_\phi\},\\
&\qquad E = \{\Phi_l(v) \mapsto F(v) \mid v \in \Sigma_\phi\}.
\end{aligned}$$

Here, $\mathit{freshNode}(v)$ denotes the value of $v$ at the entry of a given label statement fragment, where a goto statement is regarded as a control transfer into a label statement without passing through its entry.

The cases of if statements and while statements are identical to those of $\mathcal{C}$ except for placing normal $\phi$-functions instead of high-level ones. This is because an execution path via label statements do not constitute high-level control structures. For example, consider the following if statement containing a label statement:

$$\textbf{if}\ (i < n)\ \{x \leftarrow a\ \ l{:}\}\ \textbf{else}\ \{x \leftarrow b\}.$$

At the exit of the if statement above, we cannot distinguish an execution path via **goto** $l$ from an execution path of the if statement. Because of such cases, we have to place normal $\phi$-functions. The case of statement sequencing is identical to that of $\mathcal{C}$.

**$\phi$-Function Elimination** To improve the precision of equivalence detection based on congruence, we can do no better than to eliminate redundant $\phi$-functions. We can use existing methods [AH00, BBH+13] for eliminating redundant $\phi$-functions from programs in SSA form. The method by Aycock and Horspool [AH00] iterates elimination of trivial $\phi$-functions and global rewriting of variables until it converges. The method by Braun et al. [BBH+13] first find strongly connected components in the data dependency among $\phi$-functions and then eliminate them. We can eliminate redundant $\phi$-functions on value graphs by constructing equivalence classes of $\phi$-nodes. Then, $\phi$-nodes constitute union-find trees as in variable nodes. The find operation corresponds to the elimination of $\phi$-functions and rewriting of variables in both methods. The elimination of $\phi$-functions on value graphs is more efficient than that on CFGs in SSA form because we can ignore high-level $\phi$-functions in elimination. Note that both methods eliminate all redundant $\phi$-functions if a given CFG is reducible.

**Marrying with CFG-based Approach**

The standard algorithm [CFR+91] for placing $\phi$-functions is based on dominance frontiers. Dominance frontiers formalize necessary $\phi$-functions and are the most standard criterion of the minimality of $\phi$-functions. Here we calculate dominance frontiers from reduced ASTs, place $\phi$-functions, and construct value graphs from reduce ASTs.

**Construction of Reduced CFGs** A dominator tree is a tree representation of control dependence on CFGs. Dominator trees are essential for calculating dominance frontiers. Since a dominator tree is defined on a CFG, we require CFG-like structures as input. We therefore construct a *reduced CFG*, which represents the control flow of a reduced AST in the form of CFGs.

To begin with, we have to consider what point in a reduced AST becomes a node. We assign a number to a point in a reduced AST and regard the number as a node of its resultant reduced CFG. We assign numbers to join points such as the exit of an if statement, the entry of a while statement, and a label statement. Since an extended value graph represent a fork of control flow, we assign a number to each of the entry and exits of an extended value graph. We consider that each exit of an extended value graph contains the corresponding variable binding and the enter has the occurrences of free variables. Although a value graph with the empty variable binding is unnecessary for calculating dominance frontiers, dominator trees that contain value graphs as nodes are useful for constructing value graphs. We therefore assign numbers to the condition parts of an if statement and an while statement.

Next, we construct the edges of a reduced CFG. We can calculate edges, i.e., pairs of node numbers simply in a two-pass accumulative syntax-directed manner. The first pass is a bottom-up accumulation that calculate the exit node number of each subtree of a reduced AST. The first pass is a bottom-up accumulation that calculate the exit node number of each subtree of a reduced AST. The second pass is a top-down accumulation that propagate predecessor numbers. The pairs of an assigned number and a accumulated predecessor number are the edges of a reduced CFG.

Letting $n$ be the size of a reduced AST and $n_{\mathrm{J}}$ be the number of goto statements, the size (i.e., the number of nodes) of its reduced CFG is $\mathrm{O}(n+n_{\mathrm{J}})$. From the assumption of spare malignancy, this size is small. The costs of both calculating a dominator tree and calculating dominance frontiers depend only on the size of a given CFG. Both calculations on a reduced CFG are much cheaper than the standard approach based on CFGs. Both costs are negligible in the whole cost of constructing a value graph.

**Construction of Value Graphs** If a reduced CFG, its dominator tree, and its dominance frontiers are obtained, we can place $\phi$-functions in a reduced CFG by using existing CFG-based algorithms. Then, we fill the arguments of a $\phi$-functions for variable $v$ with free occurrences of $v$ (i.e., variable nodes tagged by $v$). Lastly, we connect separate (extended) value graphs by determining the values of free variables. This is a straightforward computation on the dominator tree. we start from a node of the reduced CFG in which a free occurrence of $v$, and climb up the dominator tree until $v \mapsto n \in V$ are found in a reached node. We define the value of $v$ as $n$, by unifying both nodes in value graphs. After determining the values of free variables, we obtain a resultant value graph in a form scattered over the reduced CFG. However, this scattered form does not matter because we have already constructed $\mathcal{T}$. We have successfully obtained a value graph with no redundant $\phi$-function.

## 8.5 Related Work and Discussion

The original value numbering presented by Cocke and Schwartz [CS70] was a local optimization for basic blocks. This method performs redundancy elimination on the fly while constructing value graph. A value graph limited in a basic block is a directed acyclic graph (DAG). Reif et al. [RL77, RL86, Rei78, RT82] extended value numbering to a global optimization. They used a global value graph, which is the expression DAG derived from each basic block connected with use-def chain in a given CFG. They used a birthpoint, which is a notion equivalent to $\phi$-functions but it does not form a function or an assignment. They also called a global representation of expressions a cover. Reif and Lewis [RL77, RL86] developed a method for calculating the least fixed point of covers, which is the result of copy propagation and constant

propagation/folding. Reif and Tarjan [Rei78, RT82] developed a more efficient method for calculating a simple cover, which is not optimal in the sense of the prior work [RL77, RL86] but reasonably small. The birthpoint for calculating simple covers is identical to $\phi$-functions based on dominance frontiers [CFR+91]. After the formalization of dominance frontiers had become de facto standard, the calculation of simple covers is considered as one of algorithms for placing minimal $\phi$-functions [BP03]. The method by Reif and Tarjan calculates a simple cover equivalent to SSA and applies copy propagation and constant propagation/folding to it. Lastly, by using a method by Downey et al. [DST80] for calculating congruence closures of directed graphs, it constructs the minimal simple cover, where common subexpressions are unified. Meanwhile, our method constructs value graphs through on-the-fly copy propagation. In Downey et al.'s algorithms, union-find trees were used for unify congruent subgraphs. Our choice of using union-find trees for value graphs is therefore reasonable.

Alpern et al. [AWZ88] revisited Reif et al.'s work from the perspective of SSA. They adopted $\phi$-functions for embedding control flow into value graphs. Their paper hardly described how to construct value graphs except for applying copy propagation. This is because SSA construction is equivalent to embedding value graphs into program texts. In fact, variable names are used for value numbers, i.e., the addresses of the nodes in value graphs in value numbering on SSA form [BCS97]. The technical contribution by Alpern et al. was to show the benefits of embedding high-level control structures in the form of operators into value graphs. Specifically, they introduced three operators: $\phi_{\text{if}}$ for an if statement and a pair of $\phi_{\text{enter}}$ and $\phi_{\text{exit}}$ for the entry/exit of a repeat-until statement. In later studies on SSA, these were called gating functions [BMO90].

$\phi_{\text{while}}$ used in this chapter was not an operator introduced by Alpern et al. $\phi_{\text{enter}}$ holds the depth of loop nesting to make congruence detection safe. However, as mentioned in Section 8.2, to check the containment of loops is essential. In fact, if we compare loops by their depth, only loops of the same level can become congruent. The comparison of the depth of nesting is merely an approximation. This approximation is required from the perspective of efficiency because we can implement constant-time checking of loop containment by encoding the nesting of loops into integer intervals. Although $\phi_{\text{end}}$ is also a little different from $\phi_{\text{exit}}$, both roles are the same. Use of $\phi_{\text{while}}$ and $\phi_{\text{end}}$ is not an essential difference between Alpern et al.'s work and our work.

Tate et al. [TSTL11] presented an optimization method based on equivalent transformations of program expression graphs (PEGs). Their PEGs were the same as value graphs presented by Alpern et al. except that $\phi_{\text{exit}}$ was replaced with a pair of eval/pass operators. They implemented optimizations based on algebraic transformations such as operator strength reduction by constructing augmented value graphs that contain the history of transformations. To prove the correctness of translation, They showed the pseudocode of a syntax-directed translation from the while language to a PEG. This is almost equivalent to our algorithm presented in Section 8.3. In fact, they implemented translation from a reducible CFG to a PEG [TSTL10].

It had been a folklore that the SSA construction of structured programs that have only SESE control flow such as the while language is easy. The work by Brandis and Mössenböck [BM94] was the first to clarify this folklore. They presented a one-pass syntax-directed algorithm for the SSA construction of structured programs. The idea to place $\phi$-functions described in Section 8.3 is identical to their work.

Braun et al. [BBH+13] developed a method for SSA construction from ASTs to CFGs in SSA form. Their method is seemingly similar to our work, but is actually different from ours. First of all, their method is not in a syntax-directed manner but is to perform query propagation over intermediate incomplete CFGs. Their method is a CFG-based algorithm for placing $\phi$-functions in a demand-driven manner. It implicitly prune $\phi$-functions on the basis of live variables because a query of placing $\phi$-functions propagates from use of variables backward. Their method is roughly a demand-driven version of the brute-force approach and therefore can place redundant $\phi$-functions. They also developed a method for eliminating these redundant $\phi$-functions. As mentioned in Section 8.4.3, their method for eliminating $\phi$-function is useful even for value graphs.

Our algorithms described in Sections 8.3 and 8.4.2 follow a manner of Rosen's high-level data-flow analysis [Ros77, Ros80]. Use of reduced ASTs is also a straightforward extension to Rosen's approach. Actually, extended methods [MFS79, SM14b] for taming goto/label statements utilize a notion identical to reduced ASTs explicitly or implicitly. An extension by Mintz et al. [MFS79] is to apply a CFG-based

algorithm on the fly to a smallest intermediate reduced AST. Their extension is directly applicable to our method described in Section 8.4.3. It will bring smaller reduced CFGs and thereby make it inexpensive to place $\phi$-functions on reduced CFGs. Our extension [SM14b] is inadequate for constructing value graphs because it exploits the algebraic properties of DFA as described in Section 8.4.1. Our extension utilizes reduced ASTs implicitly because it reduces the goto-free part on the fly by using Rosen's method.

## 8.6 Conclusion

In this chapter, we have presented a syntax-directed method for constructing a value graph from a given AST. Our method performs in a simple yet precise single-pass manner to the while language with break/continue (or equivalent goto/label) statements. Our method utilizes reduced ASTs for tolerating malignant goto/label statements with respect to precision or cost. The technical novelty of our method is very limited because it is based only upon existing notions and techniques. Our work, however, gives a concrete guideline for implementation. Our approach will be a basis for implementation of high-level compiler optimizations.

We have developed the prototype implementation of $\mathcal{C}$ and $\mathcal{C}_{\mathrm{A}}$ on top of COINS[7]. We also have confirmed that these are actually feasible in a realistic cost. We plan to implement value numbering by using our method and evaluate it practically. Incorporating our approach with high-level compiler optimizations is left for future work.

[7] `http://coins-compiler.sourceforge.jp/`

# Chapter 9

# Lessons from Syntax-Directed Program Analysis

We have developed syntax-directed methods for program analysis in Chapters 7 and 8. In this chapter, we examine this practice from the perspective of syntax-directed programming.

Data-flow analysis (DFA) in Chapter 7 and value-graph construction (VGC) in Chapter 8 are similar. Both formalize program fragments as functions and calculate results by composing them. This function composition is an interpretation of statement sequencing.

The production rule of statement sequencing is identical to that of *Join* in $JList_\alpha$ and both interpretations are similar. The interpretation of *Join* is the concatenation of two lists and that of statement sequencing is the sequential conjunction of two computations. It is therefore natural to interpret statement sequencing as function composition in syntax-directed program analysis. The formalization of programs with statement sequencing brings associativity useful for load balancing. This justifies our observation that formalization with trees should be first.

Our syntax-directed method for DFA can be parallelized on the basis of the independence of siblings and the associativity of function composition. We do not use segmented trees for parallelizing it because the balancing of statement sequencing suffices for the balancing of a given whole AST. In this case, a extremely deep nesting of if/while statements causes poor load balancing. Because such a nesting in computer programs is highly improbable, segmented trees are not worth using. This justifies our observation that it is important to consider the underlying data of trees.

We thus have justified our observations mentioned in Chapter 5 in this practice. Furthermore, the difference between DFA and VGC mentioned in Chapter 8 has brought an unsurprising yet important observation: true concerns in problems would not be the structures of given trees.

Recall that the difference of methods for taming goto/label statements. Its root cause is the difference of the concerns with a given program. DFA inherently requires the regular paths of a given program and does not use the control flow per se. Meanwhile, VGC inherently requires the control flow of a given program and its dominance relation for $\phi$-placement. We cannot determine the dominance relation of a program fragment in the presence of unknown control flow, while we can determine the regular paths around unknown control flow. In the presence of goto/label statements, we therefore can construct more compact intermediate results of DFA than ones of VGC. This difference between DFA and VGC is irrelevant to given trees and relevant to their problem specifications.

Our syntax-directed methods merely utilize the structure of a given tree for discovering easy parts of the whole computation. Tree computations per se are not essential. Use of trees is merely an issue on algorithmic engineering. This observation is not limited to our methods. For example, the purpose of use of search trees in various applications is acceleration, which is an issue on algorithmic engineering. From this practice, we conjecture that most of tree-based algorithms use trees for algorithmic engineering and not for problem specification.

# Part III

# Programming With Neighborhood

# Chapter 10

# Neighborhood Computations

The observations described in Chapter 5 suggest that it is unreasonable to consider from patterns of tree computations first. We should consider application domains first. In Part II, we have selected program analysis for an application domain because its computational structure is similar to list/tree skeletons. In this part, we attach importance to practical applicability and select spatial computation based on neighborhood, i.e., neighborhood computation, for an application domain.

A typical neighborhood computation is stencil computation [KBB+07, RMCKB97, BHMS91], which is extensively used in scientific computing. In the standard stencil computation, the space of a domain is divided into a uniform grid, and each cell of the grid is updated by using its neighbor cells. By mapping the grid into multidimensional arrays, the stencil computation is implemented as a regular array-based computation. This stencil computation is embarrassingly parallel because the updating of each cell. A naive parallelization is, however, insufficient in practice because this stencil computation often repeats over time steps in scientific computing. If we perform the stencil computation of each time step sequentially, it incurs a lot of cache misses because of poor locality, and results in poor performance. The existing work [KBB+07, DMV+08, TCK+11] on stencil computation therefore combined parallelization and locality enhancement.

Another typical neighborhood computation is the querying of space-partitioning trees, where a query prunes far subspaces and finds neighbors. It contains range queries to databases and nearest-neighbor queries to statistical datasets. Practical algorithms with reasonable approximation such as the Barnes-Hut algorithm for $N$-body problems and the photon mapping for global illumination utilize this querying. Because a sufficient number of independent queries are given in practice, this computation is also embarrassingly parallel. However, a naive parallel querying also has poor locality and incurs a lot of cache misses in iterative traversal of space-partitioning trees. There was also the work [JK11, JK12, JGK13] to combine parallelization and locality enhancement.

Programming for neighborhood computations thus raises the issue of locality enhancement, which is different form load balancing. Nevertheless, programming in a divide-and-conquer manner is highly appropriate because the divide-and-conquer paradigm is known to be highly advantageous both to load balancing and locality enhancement [FLPR99, FS06, BGS10]. We therefore deal with cache-efficient divide-and-conquer approaches to neighborhood computations in this part.

We first deal with stencil computation (Chapter 11). Although it is not an irregular algorithm based on trees, we investigate locality enhancement that promotes a simple divide-and-conquer computation. We secondly deal with iterative traversal of space-partitioning trees (Chapter 12). We investigate locality enhancement for both general space-partitioning trees and a skeleton that abstracts an iterative querying.

# Chapter 11

# Time Contraction for Optimizing Stencil Computation

This chapter is self-contained and is a revised and extended version of our unpublished paper [SI11b].

## 11.1 Introduction

Stencil computation appears extensively in practical applications. It appears particularly in solvers for partial differential equations (PDEs) discretized through a finite difference method (FDM). In such cases, stencil computation means to update each spatial grid point by using its temporal and spatial neighborhood. For example, the following is a typical one-dimensional three-point stencil: $x_i^{(t+1)} = c_1 x_{i-1}^{(t)} + c_2 x_i^{(t)} + c_3 x_{i+1}^{(t)}$, where $x_i$ denotes the $i$-th cell of discretized space, $x_i^{(t)}$ denotes a value of $x_i$ at the $t$-th step of discretized time, and every $c_j$ is a constant that is defined by physical conditions.

Owing to the practical importance of stencil computation, its optimization is a long-term research topic. Locality optimization [Won00, MW99, KBB+07, OG09, FS05, FS07, FS06], vectorization [HSP+11], and auto-tuning [DMV+08] for it have been investigated. Even domain-specific compilers for it have been developed [BHMS91, RMCKB97, TCK+11]. Because memory bandwidth very often becomes a major bottleneck in stencil computation, locality optimization is the most important. Time skewing [Won00, MW99], which is a specialized combination of tiling [WL91] and skewing, is the standard approach to optimizing the locality of stencil loops. Intuitively, time skewing is a reordering of computations such that a small part of the spatial grid progresses by several time-steps.

In this paper, we present a novel approach to optimizing stencil loops. First of all, notice that a stencil loop can often be represented by a recurrence equation with a linear transformation: $\boldsymbol{x}^{(t+1)} = A\boldsymbol{x}^{(t)}$, where $\boldsymbol{x}^{(t)}$ denotes a vector of spatial grid points at the $t$-th time-step, and $A$ is a sparse matrix whose non-zero elements regularly occur. For example, the following represents the above three-point stencil:

$$\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{N-1} \\ x_N \end{pmatrix}^{(t+1)} = \begin{pmatrix} c_2 & c_3 & & & \\ c_1 & c_2 & c_3 & & \\ & \ddots & \ddots & \ddots & \\ & & c_1 & c_2 & c_3 \\ & & & c_1 & c_2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{N-1} \\ x_N \end{pmatrix}^{(t)},$$

where the boundary condition is assumed to be zero-constant; i.e., $x_0 = x_{N+1} = 0$. Then, from this recurrence equation, we obtain $\boldsymbol{x}^{(t+k)} = A^k \boldsymbol{x}^{(t)}$. Our key observation is that by computing $A^k$, we can obtain a wider stencil that computes $k$ time-steps at once. That is, $k$ time-step iterations are contracted into one. However, if we compute $A^k$ and store this result naively, this contracted loop is too costly. To solve it, on the basis of the regularity of $A$, we have developed a cheap way to compute $A^k$ and a compact representation of this result. As a result, this contracted loop becomes efficient, and then it

causes fewer cache misses than the original loop since fewer time-step iterations are performed. This contraction of time-steps thus functions to eliminate redundant memory write and arithmetic, as well as to improve locality of stencil loops.

This paper makes the following contributions:

- We describe *loop contraction*, a technique to optimize a loop that recurrently applies a linear transformation, by contracting multiple iterations into one (Section 11.2). It is easy and mathematically trivial; similar formulations are found in [IKF05, LAAK06, LTA03]. However, we have not found exactly the same one in the context of loop transformations. We therefore believe that, even though not novel, it is a distinct perspective in loop optimizations.

- We have developed *time contraction*, a novel approach to optimizing stencil loops. It is a specialization of loop contraction. We describe its concept, techniques to implement it, and its effects on complexity, as well as its pros and cons (Section 11.3). Time contraction is quite different from time skewing [Won00, MW99] and is effective for reducing the cache complexity of stencil loops. This is the most significant contribution of our work.

- We have experimentally demonstrated the effectiveness of time contraction by using several implementations of a one-dimensional three-point stencil derived from the heat equation (Section 11.5). In all cases, the time contraction outperformed time skewing (i.e., the standard approach) in performance improvement: in the best settings, the time-contracted version achieved 72.4 % better performance then the time-skewed one.

- We describe tuning techniques for stencil loops on the basis of time contraction (Section 11.4). Time contraction facilities tuning of stencil loops. We have been implementing an experimental FDM library that generates tuned stencil loops by means of these techniques. In a preliminary experiment, a generated loop achieved up to 60 % better performance than a manually vectorized and time-contracted one (Section 11.5).

## 11.2 Loop Contraction: Reduction of Iterations

### 11.2.1 Explanation by Example

Consider the following loop that computes a Fibonacci number:

$$
\begin{array}{l}
x_1 \leftarrow 1 \\
x_2 \leftarrow 1 \\
\textbf{for } i = 1 \textbf{ to } n \ \{ \\
\quad \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \leftarrow \begin{pmatrix} x_2 \\ x_1 + x_2 \end{pmatrix} \\
\}.
\end{array}
$$

After this loop, $x_1$ and $x_2$ denote the $(n+1)$-th and $(n+2)$-th Fibonacci numbers, respectively. This loop can be transformed into

$$
\begin{array}{l}
\textbf{for } i = 1 \textbf{ to } n \ \{ \\
\quad \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \leftarrow \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \\
\}.
\end{array}
$$

By using loop unwinding, we obtain

$$\begin{aligned}&\textbf{for } i = 1 \textbf{ to } n/k \ \{\\ &\quad \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \leftarrow \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^k \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\\ &\}.\end{aligned}$$

For simplicity, $n$ is assumed to be a multiple of $k$. Since this matrix exponentiation is a loop-invariant expression, we can hoist it. As a result, this loop body becomes a matrix-vector multiplication. It means that iterations of the loop are contracted into one. We therefore call it *loop contraction*.

The effect of loop contraction differs from that of simply conventional optimizations for an unrolled loop. The difference is in the cost of a loop body. Consider the above example again. Since conventional optimizations (e.g., copy propagation and common subexpression elimination) do not capture the matrix exponentiation, its $k$-times-unwound conventionally optimized body necessitates $k + 1$ additions. In contrast, its $k$-times-contracted body necessitates only 4 multiplications and 2 additions of the 2-by-2 matrix-vector multiplication. The cost of a contracted body is independent of $k$; it is the core advantage of loop contraction.

If $k$ and every element of a coefficient matrix are compile-time constants like computing a Fibonacci number above, the loop-invariant matrix exponentiation can be completely computed in compile-time. In such an ideal case, loop contraction reduces the time complexity by a factor of $1/k$.

### 11.2.2 Target of Application

Stencil computation is another example to which loop contraction can be applied; it is the main target in this paper. Recall the matrix representation of the three-point stencil described in Section 11.1:

$$\boldsymbol{x}^{(t+1)} = A\boldsymbol{x}^{(t)}, \text{ where } A = \begin{pmatrix} c_2 & c_3 & & & \\ c_1 & c_2 & c_3 & & \\ & \ddots & \ddots & \ddots & \\ & & c_1 & c_2 & c_3 \\ & & & c_1 & c_2 \end{pmatrix}.$$

This is similar to the above loop that computes a Fibonacci number. Throughout this paper, we call a coefficient matrix that appears in stencil computation (like the above) a *stencil matrix* and call a vector that is iteratively updated by multiplying a stencil matrix a *grid vector*. Unless otherwise noted, $A$ and $\boldsymbol{x}$ are a stencil matrix and a grid vector, respectively. We call a contiguous non-zero part of a row of a stencil matrix a *stencil coefficient*; e.g., for the above stencil, $[c_1, c_2, c_3]$, $[c_1, c_2]$, and $[c_2, c_3]$ are stencil coefficients. In a stencil matrix, at least one stencil coefficient appears regularly and recurrently (e.g., $[c_1, c_2, c_3]$ in the above). We call such stencil coefficients *regular* ones, and call the others *irregular* ones; irregular ones represent boundary conditions for solving a governing PDE, e.g., the heat equation. Note that whereas the only regular coefficient exists near the diagonal elements of a stencil matrix in the unidimensional case, several regular coefficients are scattered in a row of a stencil matrix in the multidimensional case.

Loop contraction to a stencil loop widens its stencil; i.e., the number of non-zero elements of a stencil matrix gradually increases with loop contraction. For example, after once contraction, the three-point

stencil above is transformed into the following five-point stencil:

$$\boldsymbol{x}^{(t+2)} = A\boldsymbol{x}^{(t)}, \text{ where}$$

$$A = \begin{pmatrix} c'_{13} & c'_{14} & c'_{15} & & & & \\ c'_{22} & c'_{23} & c'_{24} & c'_{24} & & & \\ c'_1 & c'_2 & c'_3 & c'_4 & c'_5 & & \\ & \ddots & \ddots & \ddots & \ddots & \ddots & \\ & & c'_1 & c'_2 & c'_3 & c'_4 & c'_5 \\ & & & c'_{31} & c'_{32} & c'_{34} & c'_{35} \\ & & & & c'_{41} & c'_{42} & c'_{43} \end{pmatrix}.$$

This is physically natural. The $(k-1)$-time contraction means the derivation of a recurrence equation that at once computes $k$ time-steps. To capture the propagation of changes in $k$ time-steps, we require the spatially wider-area neighborhood for each grid point. In other words, loop contraction to a stencil loop is, by contracting time-steps, to transform it into an equivalent wider-time-step one.

### 11.2.3 Problem Statement

Unfortunately, the application of loop contraction to a stencil loop is useless. Recall the computation of a Fibonacci number. Let $k$ be the number of contracted iterations. The significance of loop contraction is that the cost of a contracted loop body is independent of $k$. This property is limited to the case of a dense coefficient matrix. Every stencil matrix, however, is sparse. As described above, non-zero elements in a stencil matrix increase with $k$. The cost of a contracted body hence also increases with $k$. Even though some redundancy is eliminated through contraction, in this case, loop contraction is insignificant.

The above is a problem in the aspect of time complexity. Another grave problem is in the aspect of space complexity. In the above loop computing a Fibonacci number, the size of its updated vector is 2, a quite small number. The space of a 2-by-2 matrix does not become overhead at all. In a stencil loop, in contrast, the size of a grid vector, $N$, is not a small number. The space of an $N$-by-$N$ matrix, i.e., $\mathrm{O}(N^2)$ space becomes infeasible overhead. To obtain a significant effect of loop contraction, increasing $k$ until a stencil matrix becomes a dense matrix is clearly utterly futile. Even if $k$ is small, a contracted loop necessitates more space than the original one. Consider the one-dimensional three-point stencil above. We can roughly estimate the space of the non-zero elements of a contracted stencil matrix as $\mathrm{O}(kN)$. The cost of contraction to obtain it is enormous. In contrast, the original stencil matrix necessitates only the space of $c_1$, $c_2$, and $c_3$. Loop contraction therefore worsens the space complexity of stencil loops.

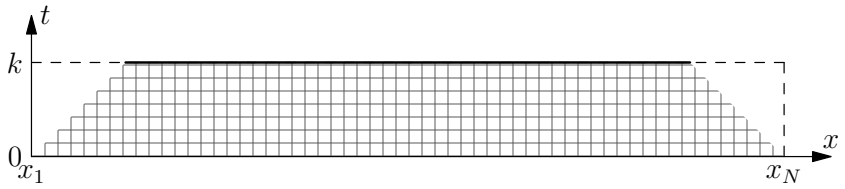
## 11.3 Time Contraction: Optimization for Stencil Loops

### 11.3.1 Our Key Observations and Solution

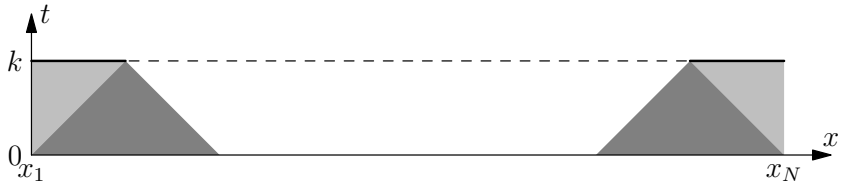
We resolve the problem stated in Section 11.2.3 and present a novel approach to optimizing stencil loops on the basis of loop contraction. The following are key observations:

- Even after modest loop contraction, a contracted stencil matrix is still a stencil one; that is, in the middle of a contracted stencil matrix, a regular pattern occurs.
- We can compute part of a grid vector by using the corresponding part of a contracted matrix.

On the basis of these observations, we have developed a specialization of loop contraction. First, we contract only the regular coefficient(s) of a stencil loop and obtain a coefficient table for its widened stencil. Then, we divide a grid vector into two parts: contracted and non-contracted. The contracted part is not influenced by boundary conditions during a contracted time-step; the non-contracted part, in contrast, is influenced. We compute the contracted part with a widened stencil. In contrast, we compute the non-contracted part in a standard manner. We repeat it at each contracted time-step. Figure 11.1 illustrates computations for the contracted and non-contracted parts in the case of a one-dimensional stencil. We call this technique *time contraction*.




(a) Computation for contracted part. Bold black line denotes result of computation. Gray-meshed part would be computed in standard approach but is skipped owing to time contraction.




(b) Computation for non-contracted part of time contraction, where $k$ time-step iterations are contracted into one. Bold black lines denote result of computation. Overall two-color gray part is computed in a standard way. Dark gray part is the intersection with gray-meshed (i.e., skipped) part in Figure a. If skipped part were computed, dark gray part would be unnecessary for computation.

Figure 11.1: Illustrations of time contraction for one-dimensional stencil, where $k$ time-step iterations are contracted into one.

**Ping-Pong Buffering** As the standard implementation of a stencil loop, two buffers for grid vectors are interleaved, as shown in Figure 11.2a. In time-contracted loops, however, interference between computations for the contracted and non-contracted parts occurs if we interleave the two buffers naively. We can avoid it easily by using an extra buffer for the neighborhood of boundaries. After computing the contracted part, we have only to interleave the extra buffer and the input one in computing the contracted part and finally store the result into the output buffer, as shown in Figure 11.2b. Then, since the non-contracted part is much smaller than the contracted part, we can use the finished part of the input buffer as the extra one. We do not have to allocate any space for ping-pong buffering. Note that if we prefer computing the non-contracted part in advance, we have only to interleave the extra buffer and the output one; then the immutate part of the output one is available for the extra one.

### 11.3.2 Precomputing Stencil Coefficients

To compute a widened stencil in a time-contracted loop, we have to prepare a coefficient table containing contracted regular coefficients. We describe our method to precompute coefficients through an example where the three-point stencil matrix described in Section 11.1 is contracted twice, i.e., the case of $A^3$.

First, we estimate the size of the time-contracted (i.e., widened) stencil. Since the original stencil is a three-point one, the size of a two-times-contracted stencil is seven. Hence, we can define $A^3$ as follows:

$$A^3 = \begin{pmatrix} * & * & * & * & & & & & \\ * & * & * & * & * & & & & \\ * & * & * & * & * & * & & & \\ r_1 & r_2 & r_3 & r_4 & r_5 & r_6 & r_7 & & \\ & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \\ & & r_1 & r_2 & r_3 & r_4 & r_5 & r_6 & r_7 \\ & & & * & * & * & * & * & * \\ & & & & * & * & * & * & * \\ & & & & & * & * & * & * \end{pmatrix},$$

where $*$ denotes a part of irregular coefficients. From the regularity of the occurrences of $[r_1, \ldots, r_7]$, we

```
for (int t = 0; t < T; ++t) {
  y[0] = x[0] + x[1];
  for (int i = 1; i < N-1; ++i) {
    y[i] = x[i-1] + x[i] + x[i+1];
  }
  y[N-1] = x[N-2] + x[N-1];
  std::swap(x,y);
}
```

(a) Naive version

```
for (int t = 0; t < T; t += k) {
  // contracted part
  for (int i = k; i < N-k; ++i) {
    y[i] = 0;
    for (int j = -k; j <= k; ++j) {
      y[i] += c[j]*x[i+j];
    }
  }
  // non-contracted part near x[0]
  for (int j = 1; j <= k; ++j) {
    z[0] = x[0] + x[1];
    for (int i = 1; i < 2*k-j; ++i) {
      z[i] = x[i-1] + x[i] + x[i+1];
    }
    std::swap(x,z);
  }
  for (int i = 0; i < k; ++i) {
    y[i] = x[i];
  }
  // non-contracted part near x[N-1]
  // is omitted, similar to the above
  std::swap(x,y);
}
```

(b) Time-contracted version, where `k` time-steps are contracted into one, `c` denotes coefficient table for its widened stencil, and `z` denotes extra buffer. For simplicity, `k % 2 == 0` is assumed.

Figure 11.2: Implementations of $x_i^{(t+1)} = x_{i-1}^{(t)} + x_i^{(t)} + x_{i+1}^{(t)}$ with ping-pong buffering, described in C++.

can see that the size of $A$ can be shrunk into a 7-by-7 matrix. Let $A_s$ be the shrunken $A$. We obtain

$$A_s = \begin{pmatrix} c_2 & c_3 & & & & & \\ c_1 & c_2 & c_3 & & & & \\ & c_1 & c_2 & c_3 & & & \\ & & c_1 & c_2 & c_3 & & \\ & & & c_1 & c_2 & c_3 & \\ & & & & c_1 & c_2 & c_3 \\ & & & & & c_1 & c_2 \end{pmatrix},$$

$$A_s^3 = \begin{pmatrix} * & * & * & * & & & \\ * & * & * & * & * & & \\ * & * & * & * & * & * & \\ r_1 & r_2 & r_3 & r_4 & r_5 & r_6 & r_7 \\ & * & * & * & * & * & * \\ & & * & * & * & * & * \\ & & & * & * & * & * \end{pmatrix}.$$

We therefore compute $[r_1, \ldots, r_7]$ as follows:

$$\begin{pmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ r_6 \\ r_7 \end{pmatrix}^T = \begin{pmatrix} 0 \\ 0 \\ c_1 \\ c_2 \\ c_3 \\ 0 \\ 0 \end{pmatrix}^T A_s A_s \iff \begin{pmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ r_6 \\ r_7 \end{pmatrix} = A_s^T A_s^T \begin{pmatrix} 0 \\ 0 \\ c_1 \\ c_2 \\ c_3 \\ 0 \\ 0 \end{pmatrix}.$$

Since $A_s^T$ is a stencil matrix, this is also stencil computation.

This algorithm is also applicable to multidimensional stencils. However, since only the occurrence of coefficients in a row of a stencil matrix is different, we have to consider shrinking stencil matrices. Consider afresh why we can shrink a stencil matrix, from a multidimensional viewpoint. That is because, to compute contracted coefficients, we require only a spatial domain into which one widened stencil fits. In other words, we shrink the spatial domain of a stencil loop to this extent and thereby its stencil matrix is also shrunk. Concretely, in the multidimensional case, we consider the bounding box of a widened stencil as the shrunk spatial domain and then obtain the corresponding shrunk stencil matrix. Note that the padding in the bounding box is convenient for computing coefficients but, of course, unnecessary to compute a stencil. Therefore, a coefficient table that holds contracted coefficients necessitates the same space as a widened stencil.

Let $S$ and $B$ be the size of the original stencil and that of the bounding box of the widened stencil, respectively. The time and space complexities of precomputing the $k$-times-contracted regular coefficients are $\mathrm{O}(kBS)$ and $\mathrm{O}(B)$, respectively. Since $S$ and $B$ are small in practice, this precomputation is very cheap. Therefore, even if we implement it as not a compile-time computation but runtime initialization, its overhead is negligible.

### 11.3.3 Advantages of Time Contraction

Time contraction is much more space-efficient than naive loop contraction, but what does it improve? To answer this question, we describe two observations:

- Loop contraction reduces memory write linearly.
- Even if a stencil is widened, we can moderate the degradation of the locality of its computation well.

The first is obviously derived from the property of loop contraction. It means a kind of bandwidth optimization because the traffic of memory bandwidth decreases. However, because memory read is dominant in a stencil loop, this effect per se might not improve the bandwidth efficiency much.

The second is much more important. It is derived from a property of stencil loops: as long as sufficient data for computing a grid point is in cache, data for computing its contiguous grid points is also in cache. Owing to this property, widening a stencil does not much increase the cache miss in computing the whole grid vector in a time-step (see also Section 11.3.4). Meanwhile, for the contracted part, time contraction reduces the number of time-steps linearly with small additional space. Computation for the non-contracted part seldom causes cache miss since it accesses a small segment of a grid vector iteratively. Time contraction thus reduces the cache miss of a stencil loop. Intuitively, time contraction transforms poor-locality computations over the time dimension into rich-locality ones over the space dimension(s). It means that time contraction is a kind of locality optimization. Because stencil loops are memory-intensive, this effect will be very significant.

It is worth noting that in a contracted time-step, each element of the output buffer is written only once. This is why we can avoid caching written data by utilizing stream writing.

Not always but usually, time contraction reduces the constant coefficient of the time complexity, i.e., arithmetic, of a stencil loop. For example, consider time contraction to the three-point stencil described in Section 11.1 where 32 iterations are contracted into one. Let $N$ be the number of grid points and $T$ be

that of time-steps. The original stencil necessitates 3 multiplications and 2 additions for a grid point; the total flop count is $5NT$. The time-contracted stencil is a 65-point one; it necessitates 65 multiplications and 64 additions for a grid point. If we ignore the non-contracted part for simplicity, the 65-point stencil is applied to $NT/32$ grid points; the total flop count is $(129/32)NT$. In this case, time contraction thus reduces about 20 % of arithmetic. This effect would be insignificant for memory-intensive computation but be significant for CPU-intensive computation. Since the improvement of the locality of a stencil loop relaxes its memory-intensiveness and makes it CPU-intensive, this effect becomes significant.

From the perspective of tuning, time contraction has two advantages. One is that time-contracted loops by nature are parametrized by $k$. It is helpful for auto-tuning. The other is that time-contracted loops have good affinities for existing techniques We describe the details in Section 11.4.

### 11.3.4 Complexity Analysis

We analyze here the effect of time contraction on the work and cache complexities of a stencil loop in the ideal cache model [FLPR99], where the work one is time complexity in the standard sense and the cache one is the number of cache misses. Note that $Z$ and $L$ denote the cache size and cache line size in the model, respectively. $N$ and $T$ denote the number of grid points and time-steps, respectively. We assume $Z \ll N$ from the motivation of locality optimization[1].

#### Unidimensional Case

We consider here a one-dimensional contiguous stencil whose size is $w$.

**Naive Loop** We consider a naive loop, which simply computes the stencil of every grid point and the whole grid vector in the same natural order. The work of a grid point is obviously $\mathrm{O}(w)$; the work of the whole grid vector is $\mathrm{O}(wN)$; the total work complexity is therefore $\mathrm{O}(wNT)$. For the cache complexity of naive loops, we assume $(\lceil w/L \rceil + 1)L \leqslant Z$. One cache miss asymptotically occurs in sequentially computing $L$ grid points. $\mathrm{O}(N/L)$ cache misses occur in sequentially computing the whole grid vector in a time-step; the total cache complexity is therefore $\mathrm{O}(NT/L)$.

**Time-Contracted Loop** We consider a time-contracted loop, where $k$ time-steps are contracted into one and $k \ll T$. $w'$ denotes the size of the widened stencil by time contraction. Since the stencil is one-dimensional, we obtain $(w-1)k + 1 \leqslant w' \leqslant wk$. We decompose the contracted part and non-contracted part in a contracted time-step.

The work complexity of a widened stencil is $cw'$, where $c$ is its constant coefficient. Since the contracted part is simply a naive wider-stencil loop, from the case of a naive loop, the work complexity of the contracted part is $cw'(N - w')$. For the non-contracted part, consider both trapezoids on the ends, in Figure 11.1b. The sum of both tops and that of both bottoms are less than or equal to $w'$ and $2w'$, respectively. Since the sum of both trapezoids is $(3/2)kw'$, the work complexity of the non-contracted part is less than or equal to $(3/2)cwkw'$. The sum of these is $cw'\{N + (3/2)wk - w'\}$. Since $(w-1)k + 1 \leqslant w'$, the work complexity in a contracted time-step of the whole vector is $\mathrm{O}(w'N + w'^2)$. If $w' \ll N$, $\mathrm{O}(w'N + w'^2) = \mathrm{O}(w'N)$. In this case, the total work complexity is therefore $\mathrm{O}(w'NT/k) = \mathrm{O}(wNT)$.

For the cache complexity, we consider only the case of $(\lceil w'/L \rceil + 1)L \leqslant Z$ since the cache complexity obviously increases in the case of its negation. The cache miss in a contracted time-step of the contracted part is asymptotically $\mathrm{O}(N/L)$. That of the non-contracted part is asymptotically $\mathrm{O}(w'/L)$ if $2w' \leqslant Z$. In this case, since $w' \ll N$, the sum of these is $\mathrm{O}(N/L)$. The total cache complexity is therefore $\mathrm{O}(NT/Lk)$ if $2w' \leqslant Z$.

**Summary** In the unidimensional case, time contraction does not change the work complexity asymptotically but does improve the cache complexity by a factor of $k$. Through the analysis above, we obtain

[1] In the case of $Z \gtrapprox N$, the cache miss is asymptotically negligible.

two observations: one is that the complexity of the non-contracted part is asymptotically negligible; the other is that time contraction roughly changes complexity by replacing $w$ with $kw$ and $T$ with $T/k$.

#### Multidimensional Case

We now sketch the analysis of the $d$-dimensional case. Contiguous data over the leading dimension has contiguous memory addresses. For convenience, we assume that the spatial domain is a hypercube whose side is $N^{1/d}$, consider a $d$-cube stencil whose side is $w$, and consider only the case of $w^{d-1}(\lceil w/L \rceil + 1)L \leqslant Z$.

The work complexity does not matter; similarly, time contraction does not worsen it asymptotically. Meanwhile, the cache complexity matters. On the basis of the two observations obtained in the unidimensional case, we analyze only the cache complexity of the contracted part and roughly estimate the effect of time contraction by replacing $w$ with $kw$ and $T$ with $T/k$ in complexity.

**Naive Loop** The computation of contiguous $L$ grid points causes $w^{d-1}$ cache misses. One computation from end to end on the leading dimension causes $\mathrm{O}(w^{d-1}N^{1/d}/L)$ cache misses. The computation in a time-step of the whole grid vector causes $\mathrm{O}(w^{d-1}N/L)$ cache misses. The total cache complexity is $\mathrm{O}(w^{d-1}NT/L)$.

Time contraction roughly changes $w$ into $kw$ and $T$ into $T/k$. The application of time contraction to a naive stencil loop changes the cache complexity into $O(k^{d-2}w^{d-1}NT/L)$ if $w^{d-1}(kw + L) \leqslant Z$. Hence, in the multidimensional case (i.e., $d \geqslant 2$), time contraction to a naive stencil loop does not improve its cache complexity.

**Hypercube Blocking** Consider $d$-dimensional blocking, where the block is a $d$-cube whose side is $\beta$. The computation of a block necessitates $(\beta + w)^d$ space. If $(\beta + w)^d \leqslant Z$, the computation of a block asymptotically causes $\mathrm{O}((\beta + w)^{d-1}\beta/L)$ cache misses, in space-filling-curve traversal among blocks. A grid vector consists of $N\beta^{-d}$ blocks. The cache complexity of the computation in a time-step is $\mathrm{O}(N\beta^{-d}(\beta+w)^{d-1}\beta/L) = \mathrm{O}((1+w/\beta)^{d-1}N/L)$. The total cache complexity is $\mathrm{O}((1+w/\beta)^{d-1}NT/L)$.

For the blocked stencil loop, time contraction roughly changes the cache complexity into $\mathrm{O}((1 + kw/\beta)^{d-1}NT/Lk)$ if $(\beta + kw)^d \leqslant Z$. Hence, as long as selecting $k$ and $\beta$ such that $kw \ll \beta \leqslant Z^{1/d} - \beta$, time contraction to a blocked stencil loop reduces cache complexity by a factor of $1/k$.

**Summary** In the multidimensional case, time contraction also improves the cache complexity by a factor of $k$ under appropriate parameter settings. However, constraints on $k$ and $Z$ are qualitatively severe with the increase of $d$, and unlike the unidimensional case, we require to perform blocking over the space dimensions carefully on the basis of $Z$ and $N$ together. Although spatial blocking has priority over time contraction, time contraction is orthogonally effective.

### 11.3.5 Extension and Applicability

We have dealt with stencil loops in the form $\boldsymbol{x}^{(t+1)} = A\boldsymbol{x}^{(t)}$. However, this is not very practical. As an example of PDEs, consider the heat equation. If we can use only the form $\boldsymbol{x}^{(t+1)} = A\boldsymbol{x}^{(t)}$, we can give only initial and boundary conditions; we cannot give external input, e.g., heat sources. To deal with external input, we need to use the form $\boldsymbol{x}^{(t+1)} = A\boldsymbol{x}^{(t)} + \boldsymbol{b}$, where $\boldsymbol{b}$ represents heat sources that emit constant quantities of heat at each time-step. For the form $\boldsymbol{x}^{(t+1)} = A\boldsymbol{x}^{(t)} + \boldsymbol{b}$, in fact, time contraction is also applicable. By expanding the induction, we obtain the following equation:

$$\boldsymbol{x}^{(t+k)} = A^k\boldsymbol{x}^{(t)} + \bar{A}\boldsymbol{b}, \text{ where } \bar{A} = \sum_{i=0}^{k-1} A^i.$$

Here, $\bar{A}$ is also a stencil matrix. Like $A^k$, we require only the regular part of $\bar{A}$. We can compute $\bar{A}$ at the same cost as $A^k$ by accumulating the intermediate results of $A^k$. After we obtain $\bar{A}$, we can compute $\bar{A}\boldsymbol{b}$ in almost the same cost as that of only one time-step iteration of a time-contracted loop. Even though $\bar{A}\boldsymbol{b}$ is usually more expensive than $A^k$ and $\bar{A}$, compared to the overall cost, this precomputation is still

negligible overhead. Once we compute $A^k$ and $\bar{A}\boldsymbol{b}$, the contracted form above becomes the same as the original form; loop contraction thus succeeds.

The fact that time contraction is applicable to the form $\boldsymbol{x}^{(t+1)} = A\boldsymbol{x}^{(t)} + \boldsymbol{b}$ implies that it is also applicable to solving $A\boldsymbol{x} = \boldsymbol{b}$ through the Jacobi method. In the relaxation of the Jacobi method, $\boldsymbol{x}$ is iteratively updated; $\boldsymbol{x}^{(t+1)} = D^{-1}\boldsymbol{b} - D^{-1}R\boldsymbol{x}^{(t)}$, where $D$ is the diagonal component of $A$ and $R$ is the remainder. If $A$ is a stencil matrix, $D^{-1}R$ is also a stencil matrix. Here, $t$ does not physically mean time, but we can regard the relaxation process as the progression of time. In fact, solving $A\boldsymbol{x} = \boldsymbol{b}$ corresponds directly to solving Poisson's equation through an FDM. In addition, solving $A\boldsymbol{x} = \boldsymbol{b}$ produces another application. The standard form $\boldsymbol{x}^{(t+1)} = A\boldsymbol{x}^{(t)}$ is actually derived from an explicit FDM, which is one of the simplest FDMs. In contrast, the forms $A\boldsymbol{x}^{(t+1)} = \boldsymbol{x}^{(t)}$ and $A\boldsymbol{x}^{(t+1)} = A'\boldsymbol{x}^{(t)}$ are derived from an implicit FDM and the Crank-Nicolson method respectively, which are more precise and numerically stable than an explicit FDM. These methods necessitate solving $A\boldsymbol{x} = \boldsymbol{b}$ at every time-step. We thus can apply time contraction to it.

Worth noting is that time contraction is applicable to the finite domain time difference (FDTD) method, which is the standard FDM for Maxwell's equations in electromagnetism, and the lattice Boltzmann's method (LBM), which is the standard discretization scheme for computational fluid dynamics and whose implementation is a part of SPEC 2006. Both are the stencil computation in the form $\boldsymbol{x}^{(t+1)} = A\boldsymbol{x}^{(t)}$. For example, by pairing the electric field $\mathbf{E}$ and magnetic field $\mathbf{H}$ at the same point, the one-dimensional FDTD is known to be converted into a three-point stencil [OG09]. The standard two-dimensional and three-dimensional LBMs are implemented as a 9-point stencil and 19-point one.

Although we can apply time contraction to the stencil computation derived from various discretization schemes as described above, we cannot apply it to every stencil computation. There are three essential limitations. The first is that it cannot be applied to the Gauss-Seidel method for solving $A\boldsymbol{x} = \boldsymbol{b}$. This is because the product of a stencil matrix and a vector cannot be extracted. The second is that time contraction is almost useless for dynamic programming such as seam curving [AS07]. To contract iterations is to avoid computing intermediate results. If we have to store all intermediate results into a memo table, time contraction as well as loop contraction is useless. However, if we can compute a solution from skipped intermediate values efficiently, time contraction can be effective. The third is that time contraction is not effective for problems with boundaries in the middle of the spatial domain, e.g., the case in the heat equation where insulation objects exist in the middle. The efficiency of time-contracted loops and its precomputation owes much to the regular occurrences of stencil coefficients. The existence of boundaries in the middle disrupts this regularity. We can deal with boundaries in the middle by well segmenting the non-contracted part, but its implementation becomes non-trivial.

### 11.3.6 Drawback: Lowering Precision

Since time contraction performs algebraic transformation on finite-precision floating-point numbers, it changes floating-point errors. Unfortunately, in practical situations, time contraction has a negative effect on loss of significance.

Consider, for example, the one-dimensional heat equation discretized through an explicit FDM. We obtain the following symmetric three-point stencil: $x_i^{(t+1)} = rx_{i-1}^{(t)} + (1-2r)x_i^{(t)} + rx_{i+1}^{(t)}$, where $r \leqslant 1/2$ for numerical stability. Hence, each element of the regular stencil coefficient is always less than 1. Because the heat equation is a diffusion equation, it is natural that the sum of all elements of the regular stencil coefficient is 1. Note that this property of regular coefficients is common in other PDEs. As a result, the variance of the elements of $A^k$ sharply increases with $k$. Figure 11.3 illustrates the values of the regular stencil coefficient of the three-point stencil above in the case of $r = 1/3$ and their changes with increasing $k$. Qualitatively, loss of significance by a factor of the vertical distance between two adjacent points on the same line in Figure 11.3 is caused. Quantitative loss of significance depends on the values of grid points. Even if we compute a stencil inward from both ends, the order of each intermediate result little changes. Therefore, this loss of significance is, unfortunately, almost unavoidable. Time contraction essentially has such a performance-precision trade-off.

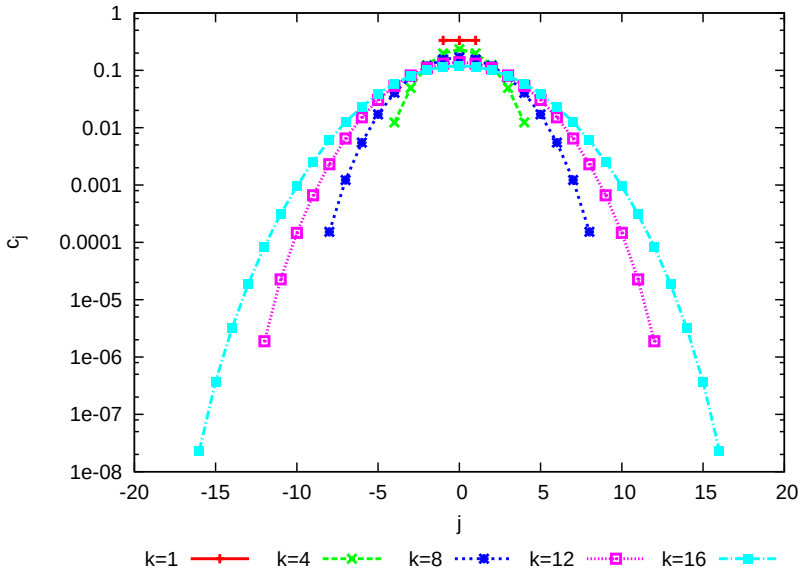



Figure 11.3: Change of regular coefficients with contraction, where stencils are $x_i^{(t+k)} = \sum_{j=-k}^{k} c_j x_{i+j}^{(t)}$.

## 11.4 Tuning Based on Time Contraction

### 11.4.1 Parametricity on $k$

A parameter $k$, the number of iterations contracted into one is the primary parameter of time contraction. Time contraction improves locality over the time dimension by a factor of $k$. Coefficient tables, however, increase by a factor of $k$, and the benefits of time contraction premises modest $k$. Tuning $k$ to an appropriate scale is thus necessary in time contraction.

As can be seen from Figure 11.2b, time-contracted loop by nature are parametrized by $k$. In other words, even though $k$ is a dynamic value, the form of every time-contracted loop does not change at all. Owing to this parametricity, we can tune $k$ dynamically and adaptively. Although the code of time-contracted loops is parametric, the data of their coefficient tables is dependent on $k$. When we change $k$, we also have to update a coefficient table. As described in Section 11.3.2, the construction of coefficient tables can be implemented as recurrent computation over $k$. Consequently, when we increment $k$ monotonically, we can construct coefficient tables incrementally. Of course, when the upper bound of $k$ is given, we can prepare all coefficient tables in advance. Time contraction is therefore well-suited to auto-tuning.

### 11.4.2 Affinity for Unroll-and-Jam

The main computational kernel of a time-contracted loop is, of course, the computation for its contracted part. As can be seen from Figure 11.2b, the structure of this kernel is simple and compact. It leads to an advantage of time contraction. This computational kernel has a good affinity for unroll-and-jam [AK01].

Unroll-and-jam is an optimization for a nested loop; it is to perform unrolling of an outer loop followed by fusion of duplicated successive inner loops. For example, by applying it to the loop for the contracted part in Figure 11.2b, we obtain a loop shown in Figure 11.4. By using unroll-and-jam, we can vectorized the innermost loop for the contracted part easily. Then, usage of the next grid vector (i.e., `y`) becomes efficient by employing scalar replacement. Unroll-and-jam promotes usage of registers but complicates the body of the innermost loop. An excessively complicated body causes spilling and can degrade performance. Meanwhile, time contraction increases the number of iterations of the innermost loop but does not complicate its body. Time contraction thus does not worsen its disadvantage and promotes its advantage. While time contraction improves cache-level locality, unroll-and-jam improves

```
for (int i = k; i < N-k; i += 2) {
  y[i+1] = y[i] = 0;
  for (int j = -k; j <= k; ++j) {
    y[i]   += c[j]*x[i+j];
    y[i+1] += c[j]*x[(i+1)+j];
  }
}
```

Figure 11.4: Example of unroll-and-jam; original loop is the loop for the contracted part in Figure 11.2b.

```
float **x, **y; // N-by-N spatial grid
int h; // maximum depth of recursion
const int L = N-2*k;
for (long bv = 0; bv < 4<<h; ++bv) {
  // calculate position of a base block
  int si = 0, sj = 0;
  for (int i = 0; i < h; ++i) {
    si += (bv & 0x1<<2*i) * (L>>h<<i);
    sj += (bv & 0x2<<2*i) * (L>>h<<i);
  }
  // compute a base block
  for (int i = si; i < si+(L>>h); ++i)
    for (int j = sj; j < sj+(L>>h); ++j)
      y[i][j] = compute_stencil(x,i,j);
}
```

Figure 11.5: $h$-level divide-and-conquer blocking for contracted part of 9-point square (two-dimensional) stencil.

register-level locality. Tuning of both leads to near-ideal performance.

### 11.4.3 Affinity for Divide-and-Conquer

It is known that the divide-and-conquer approach is useful for locality optimization of stencil computation. Cache-oblivious algorithms for stencil computation [FS05, FS07, FS06] suppresses cache misses on the basis of the divide and conquer of iteration space. Such a divide-and-conquer technique, however, complicates the structure of stencil computation. Time contraction simplifies this complication.

In an cache-oblivious algorithm for a one-dimensional stencil, for example, a computational trapezoid (corresponding to the meshed trapezoid in Figure 11.1a) is divided into two parts and recursively computed in an appropriate order. Then, the two divided parts in the same recursive call are different in shape and size; moreover, there is dependence between them. These are obvious disadvantages in parallelizing stencil computation into a load-balanced form. The cause of these is the slope of the trapezoid. Recall that in Figure 11.1a, only the top of the trapezoid is computed. The computation for the contracted part does not have any slope. Consequently, by applying the divide and conquer over only the space dimension to the contracted part, we obtain a regular recursion easier to parallelize and to load-balance.

The divide-and-conquer approach is effective especially for multidimensional stencils. As mentioned in Section 11.3.4, in optimizing locality of multidimensional stencils, blocking over the space dimensions is necessary in practice. The divide-and-conquer approach is well-suited to this. Then, the depth of recursion is a good parameter on task granularity. The maximum depth of recursion, $h$, is easier to tune than the block size itself in simple blocking. Furthermore, we can generate simple loops parametric on $h$ with bitwise operations, as shown in Figure 11.5.

## 11.5 Experiments

### 11.5.1 Experimental Library

We have been implementing an experimental FDM library that generates tuned stencil loops on the basis of the approaches described in Section 11.4. Our library takes the specification of a stencil and tuning parameters, constructs a coefficient table, and generates a tuned stencil loop in C++. Generated loops are vectorized and tuned on usage of registers. In addition, by employing OpenMP directives, they support multithreading.

Our library is under development. Since the current version is very restrictive, its implementation itself is too weak to be a contribution of this paper. It is a testbed of tuning techniques.

### 11.5.2 Experimental Settings

**Problem** We conducted experiments on the one-dimensional three-point stencil derived from the discretization of the heat equation by means of the standard explicit FDM described in Section 11.3.6. We performed a simulation of thermal diffusion in an object whose initial temperatures in one half and the other were 800 and 300 respectively; both of whose ends were insulated. From properties of the simulation object, we defined $r = 0.1526$.

This stencil is a relatively simple one but is appropriate to the comparison between time contraction and time skewing. It is because in unidimensional stencils, locality on the space dimension does not matter. Time contraction alone improves only locality on the time dimension. To improve locality on the space dimensions in multidimensional stencils, we have to employ other techniques, such as divide-and-conquer blocking described in Section 11.4. In addition, as described in Section 11.4, time contraction does not disturb blocking. To evaluate effects of time contraction itself, unidimensional stencils are therefore sufficient.

**Programs** We implemented by hand four versions: `ref` is a reference implementation with no manual optimization, `vec` is a vectorized one, `tiled` is a vectorized and time-skewed one, and `contrd` is a vectorized and time-contracted one. We implemented every vectorized version by using intrinsic functions. We employed the standard vectorization scheme with the `palingr` instruction of SSSE3, described in [HSP+11]. To every vectorized version, we had added OpenMP directives; it thus was able to perform multithreading optionally. `tiled` was implemented with split tiling [KBB+07], and its parallelization scheme was the same as described in [KBB+07]: computing all tiles in parallel and then computing all the residual parts in parallel. In addition to these manual versions, we used `gen`, a version generated by our library. Every floating-point number used in these programs was `float`.

**Environments** We used two different environments. For single-threaded programs, we used a machine equipped with Core 2 Duo E8500 (2 cores; 3.16 GHz) and 4 GB (DDR2-800) running Linux 2.6.31 (32-bit). Its ideal performance per core is 12.64 GFLOPS[2]. For multithreaded programs, we used a machine equipped with Xeon X5550 (4 cores × 2; 3.06 GHz) and 12 GB (DDR3-1333) running Linux 2.6.38 (64-bit). Its ideal performance is 98 GFLOPS[3]. For both processors, Hyper-Threading was disabled. Each single-threaded program was compiled by g++ 4.4.1; each multithreaded one was compiled by g++ 4.5.2. The default optimization level of g++ was `O3`.

### 11.5.3 Experimental Results

In this section, $N$ and $T$ are the same as described in Section 11.3.4; $B_t$ denotes the number of grid points updated per tile; $k$ denotes, in time skewing, the block size of time-steps, and in time contraction, the number of time-steps contracted into one. In addition, $u$ denotes the count of unroll-and-jam for `gen`. Note that `gen` where $u = 1$ corresponds approximately to `contrd`. The FLOPS of `contrd` and `gen`

[2] http://www.intel.com/support/processors/sb/CS-020870.htm
[3] http://download.intel.com/support/processors/xeon/sb/xeon_5500.pdf

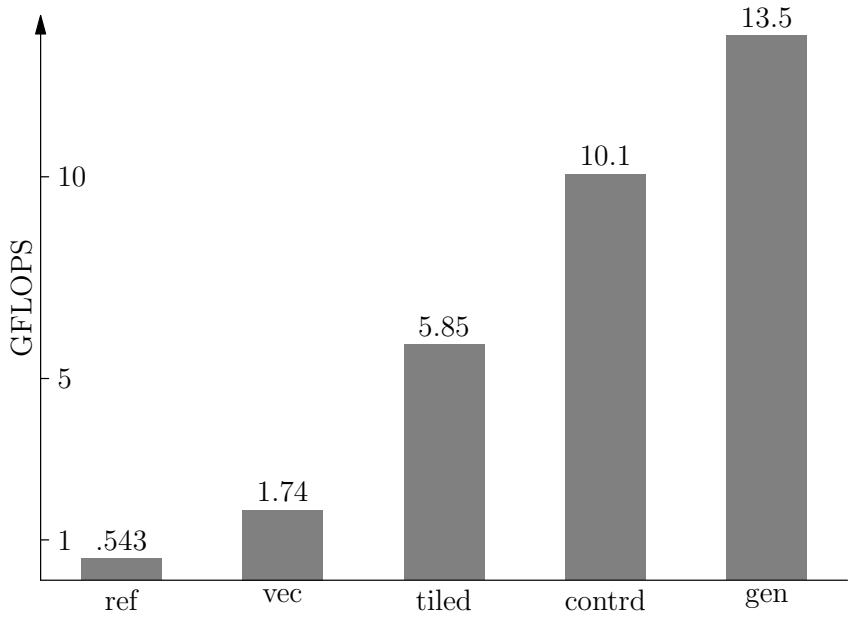



Figure 11.6: General single-threading performance; $N = 16$ Mi, $B_t = 4$ Ki, $k = 32$, and $u = 7$.

is based on the flop count of `ref`. Since time contraction reduces arithmetic (see also Section 11.3.3), these performance can outstrip the ideal performance of the environments.

**General Performance** Figure 11.6 illustrates the performance of each version in our best sequential settings. `contrd` outperformed `tiled`: `contrd` achieved 479 % better performance than `vec` and 72.4 % better performance than `tiled`. Since `contrd` and `tiled` are sufficiently locality-optimized, we consider that the difference between `contrd` and `tiled` originated at the reduction of memory write and arithmetic by contraction. Meanwhile, the difference between `contrd` and `gen` was caused by the improvement of usage of registers: `gen` achieved 34.2 % better performance than `contrd`.

**Effects of $k$** Both time contraction and time skewing improve locality by a factor of $k$. Figure 11.7 illustrates the effects from scaling $k$. `contrd` always outperformed `tiled` for the same $k$. Moreover, whereas for `tiled` we have to tune $k$ and $B_t$, for `contrd` we have only to tune $k$. From these results, we consider that time-contracted loops are easier to tune than time-skewed ones. Note that in the case of $k > 32$, we observed no improvement in performance for every version. We consider that for every version, the locality improvement reached a limit in $12 \leqslant k \leqslant 16$ and the performance improvement in $16 \leqslant k \leqslant 32$ was caused by constant factors.

**Effects of Unroll-and-Jam** Our library performs code generation by means of miscellaneous techniques. The most effective technique for improving usage of registers was unroll-and-jam. Figure 11.8 illustrates the effects of unroll-and-jam. In all of these cases, unroll-and-jam provided significant speedup. Overall, unroll-and-jam was more effective for the case of large $k$. It has demonstrated a good affinity between time contraction and unroll-and-jam.

**Multithreading** As locality improves, the effect of parallelization generally becomes significant. Figure 11.9 illustrates the effects of multithreading of the stencil loops. Note that the environment of this experiment is different from that of the others. Whereas `vec` gained no improvement from multithreading, `tiled` and `contrd` achieved near-linear scalability. As can be seen from this result, memory-intensiveness in `contrd` as well as `tiled` was sufficiently relaxed. Furthermore, the scalability of `contrd` was always

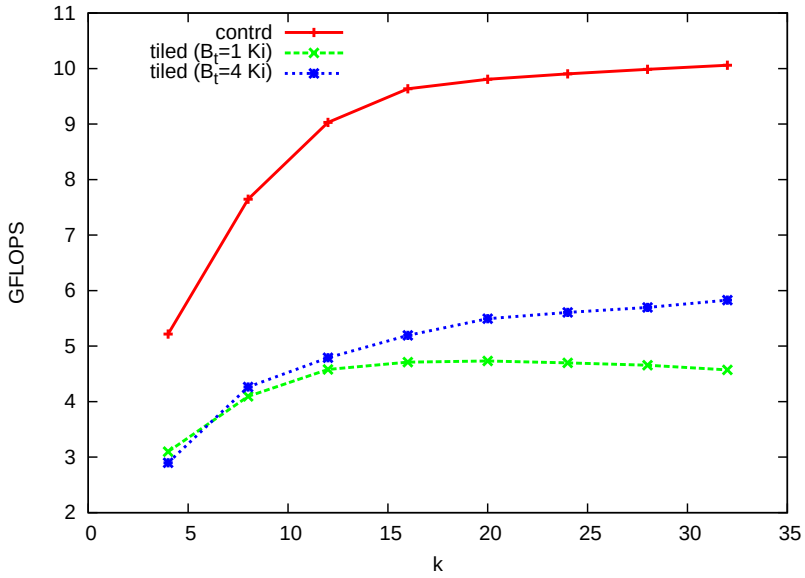



Figure 11.7: Effects from scaling $k$; $N = 16$ Mi.

closer to the linear than that of `tiled`. The unroll-and-jam in multithreading was more effective than that in single-threading: `gen` achieved up to 61.1 % better performance than `contrd`.

**Errors** As described in Section 11.3.6, time contraction causes more loss of significance as $k$ increases. Figure 11.10 illustrates the geometric mean of relative errors of `contrd` in the case where the analytic solution is $\exp(-\pi^2 t)\sin \pi x$. The result that `contrd` was more precise than `ref` in all the cases contradicted our prediction. We consider that this contradiction originated at precomputing stencil coefficients in `double`. At least, compared to the default errors, loss of significance caused by time contraction was negligible.

**Summary** In all the experimental results, `contrd` outperformed `tiled` in the corresponding settings. We, however, do not absolutize this superiority of time contraction in performance for every stencil loop. As mentioned above, the performance improvement in unidimensional stencils is the genuine effect of time contraction. Since blocking over the space dimensions is important for multidimensional stencils (as described in Section 11.3.4), the time contraction to multidimensional stencils would be less effective than that to unidimensional stencils. As a result, the superiority of time contraction to time skewing would be a little. However, it means that locality on the time dimension becomes relatively less important; it is not a disadvantage of time contraction per se. What we allege is that time contraction is sufficiently comparable to time skewing and outstrips time skewing in the genuine effect on performance. The experimental results have supported our allegation.

## 11.6 Discussion

We have presented time contraction, a novel approach to optimizing stencil loops, and demonstrated its effectiveness. In this section, as concluding remarks, we discuss the connections and differences between our work and existing work, and finally discuss future work.

### 11.6.1 Related Work

**Optimization for Stencil Computation** Since stencil computation is practically important, there are many studies focusing on its optimization. In earlier studies, domain-specific compilers for stencil

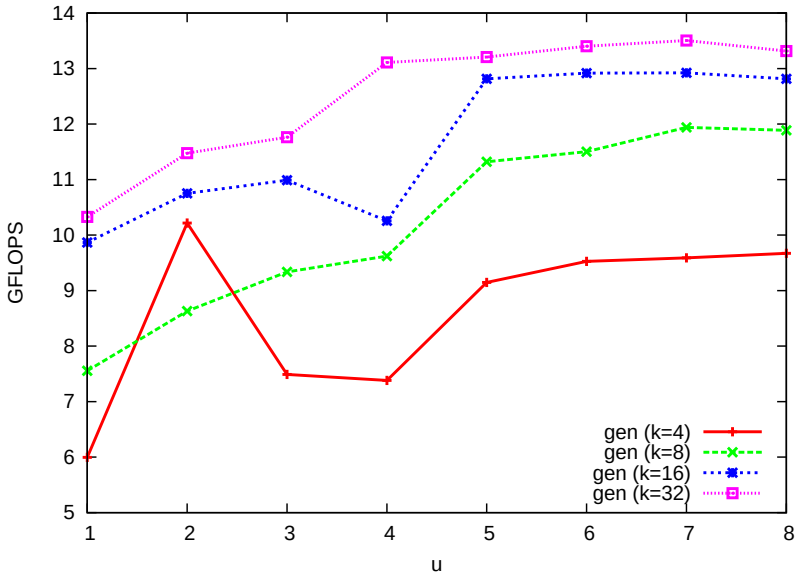



Figure 11.8: Effects from scaling $k$ and $u$; $N = 16$ Mi.

computation were developed, e.g., one for the connection machine CM-2 [BHMS91] and HPF's one [RMCKB97]. In the most recent study, Henretty et al. [HSP+11] have presented a vectorization technique specialized for stencil computation.

Tiling [WL91] is the standard technique to improve the locality of general nested loops. McCalpin and Wonnacott [MW99, Won00] presented time skewing, a combination of tiling and skewing specialized for stencil loops. Today, time skewing (or similar skewed tiling techniques) has become the standard approach to optimizing stencil loops [KBB+07, DMV+08, OG09, BHT+10]. Krishnamoorthy et al. [KBB+07] presented overlapped and split tilings for well load-balanced parallelization. Orozco and Gao [OG09] improved split tiling as diamond-shape tiling for FDTD on the Cyclops-64 many-core chip architecture. Datta et al. [DMV+08], by using tiling as well as hardware-aware optimizations such as prefetching, developed an auto-tuning environment for stencil computation. Baskaran et al. [BHT+10] developed parallel parametric tiling for auto-tuning; note that their work is well-suited for but not limited to stencil loops.

Frigo et al. [FLPR99] presented the cache-oblivious algorithm, whose algorithmic parameters are independent of hardware parameters in contrast to hardware-aware techniques such as Datta et al.'s ones. They developed cache-oblivious algorithms for stencil computation [FS05, FS07, FS06], which are based on a divide-and-conquer approach over iteration space. In the most recent study, on the basis of these algorithms, the Pochoir stencil compiler [TCK+11] has been developed.

As far as we know, all of the existing work on the optimization for stencil computation is based only on the reordering of computations for grid points. The concept and techniques of time contraction are quite different from that; it is based on the derivation of a wider stencil. Nevertheless, in hindsight, time contraction can be seen as simplified overlapped tiling. When we narrow an overlapped trapezoidal tile, each tile becomes a triangle (as illustrated in Figure 11.11); then, by contracting each triangle in the time direction, we obtain computation of the contracted part. Therefore, it is natural that time contraction reduces memory write and arithmetic as well as improving locality.

**Work Related to Loop Contraction** We can find formulations similar to loop contraction in the context of recursion removal, such as in work by Ichikawa et al. [IKF05] and Luca et al. [LAAK06]. While both have differences in the target and approach, both extract from a recursion the form of $A^n \boldsymbol{v}$, where each size of $A$ and $\boldsymbol{v}$ is a constant and $A$ denotes a matrix of closed-form expressions. Then, $A^n$ is computed in $\mathrm{O}(\log n)$ time. Loop contraction deals with almost the same form, but there are

two differences: 1) a given program is iterative and 2) $A^n$ is partially computed in advance, especially in compile-time. In essence, whereas their work is oriented to algorithmic change, loop contraction is oriented to partial evaluation. The difference is only in the focus.

In the context of loop optimization, Lamb et al.'s work [LTA03] resembles loop contraction. The target of their optimization is a stream program that applies a linear transformation to input streams. Their optimization operates the pipeline and join of streams as operations on transformation matrices. In other words, it performs loop fusion on transformation matrices. Unlike loop contraction, it does not reduce the number of iterations,

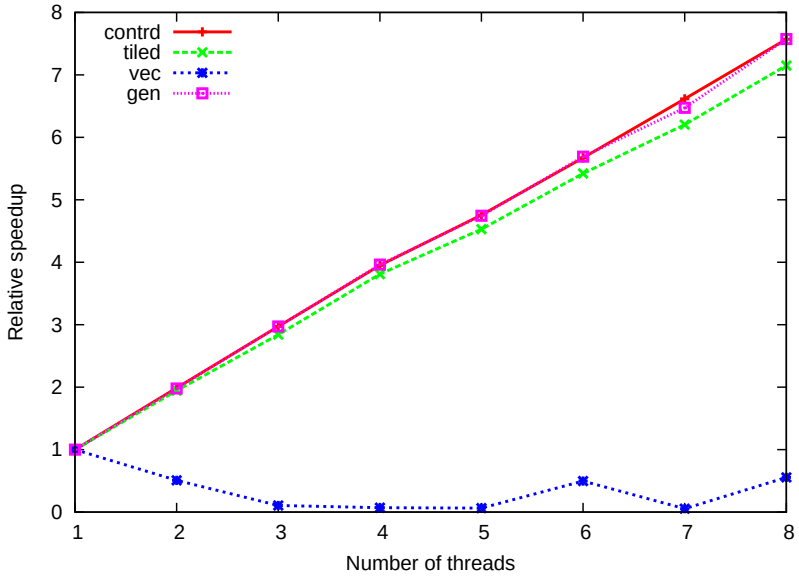
Of course, loop contraction is only a combination of unroll-and-jam and loop-invariant code motion (as well as constant folding), specialized for linear transformations. We therefore claim neither its novelty nor difficulty. The novelty of our work is time contraction, a specialized application of loop contraction to stencil loops. Since time contraction owes much to the property of a stencil matrix, it is difficult to implement only with a simple combination of existing techniques.

Nevertheless, the improvement of time contraction over loop contraction is not difficult; its idea and technique themselves seem even trivial. We therefore consider that the novelty and significance of time contraction is on its perspective rather than its technique.

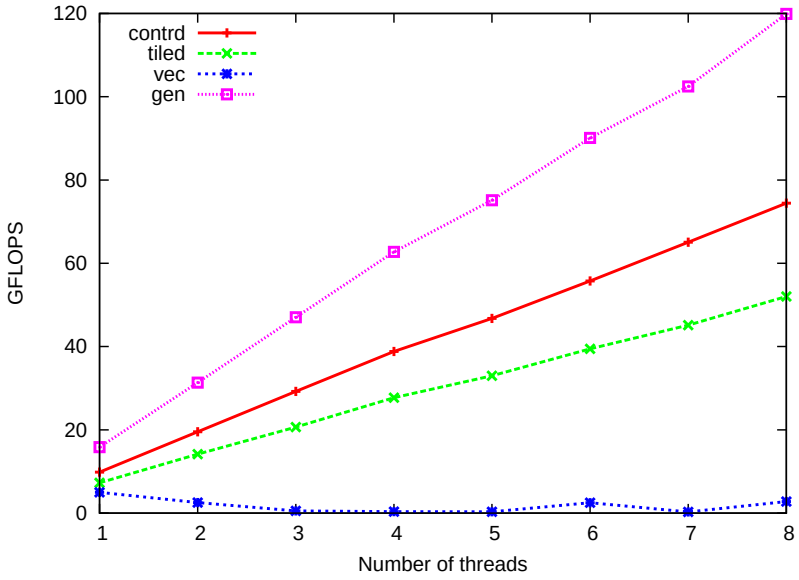
### 11.6.2 Future Work

**Implementation** We have implemented time contraction only into an experimental FDM library. We believe that this approach is cost-effective. Meanwhile, we consider that there is no technical difficulty in implementing time contraction to simple stencils in a compiler because the analysis for tiling is also sufficient for time contraction. However, implementation of time contraction to complicated stencil loops–especially dealing efficiently with boundaries in the middle–is nontrivial. We guess that rather than a precise dependence analysis, a symbolic execution is well-suited to it. We leave it for future work.

**Evaluation** The drawbacks of time contraction can be summarized as two points: the limitations in applicability (Section 11.3.5) and the increase of loss of significance (Section 11.3.6). Evaluation on these points as well as speedup through practical benchmarks is important. In particular, whether the limitation on boundaries matters in practical situations is quite important. The practicality of time contraction strongly depends on this.




(a) Relative speedup.




(b) Absolute performance.

Figure 11.9: Multithreading performance; $N = 16$ Mi, $B_t = 4$ Ki, $k = 32$, and $u = 9$.

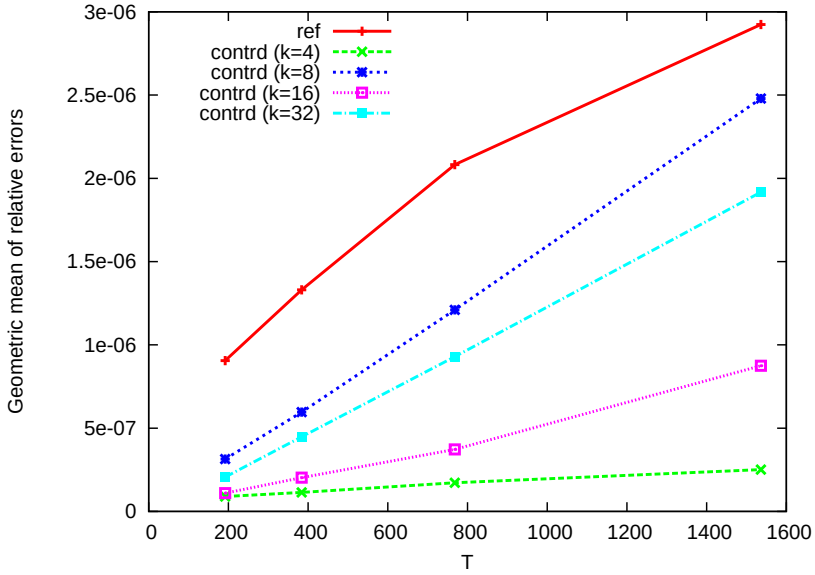



Figure 11.10: Geometric mean of relative errors of `contrd` and `ref` in scaling $k$ and $T$; $N = 1$ Ki.

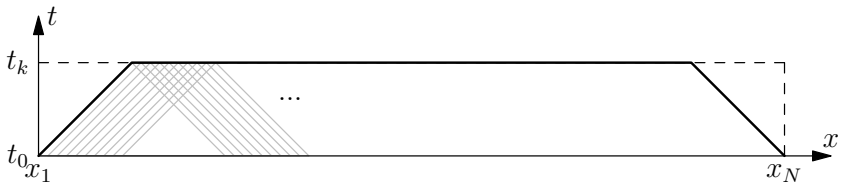



Figure 11.11: Finest-grained overlapped tiling. If each triangle tile is contracted in the time direction, this becomes Figure 11.1a.

# Chapter 12

# Locality Enhancement based on Segmentation of Trees

This chapter is self-contained and is a revised and extended version of our unpublished paper [Sat14a].

## 12.1 Introduction

A common way of searching fast for points in space is to use a space-partitioning tree such as octree, $k$-d tree, and vantage-point tree. In addition, common algorithms in important applications such as $N$-body simulation, ray tracing, and machine learning iterate a traversal of a space-partitioning tree for a given set of querying points. Although there are several variations in traversal pattern, each iteration of tree traversal is independent among query points. Such iterative tree traversal is therefore easy to parallelize.

Although such parallelization enables us to utilize multiple processors, it cannot avoid a lot of cache misses because of noncontinuous memory access in tree traversal. Parallel processing is necessary for higher performance but not sufficient. Even though multiple processors are available, they share one memory bus. A lot of cache misses occupy memory bandwidth and make all processors stall. To obtain higher performance of iterative tree traversal, we require improving its locality and reducing cache misses as well as parallelizing it.

One way of reducing cache misses in tree traversal is to use existing cache-efficient/-oblivious tree data structures [AADHM03, BDFC05, BFCF$^+$07, BKTW11]. By using them, we can reduce cache misses in each traversal. However, because iterative tree traversal is a common computational pattern, to apply both locality enhancement and parallelization simultaneously to the whole computation is reasonable and promising for efficient implementation. An abstraction of iterative tree traversal incorporated with both locality enhancement and parallelization is therefore desirable.

As an abstraction of tree computation incorporated with parallelization, tree skeletons [Ski96, GCS94] were studied. In the implementation of tree skeletons, the $m$-bridging technique [Rei93], which is to decompose a tree into tree segments, was used for distributed-memory computation [Mat07a]. Although $m$-bridging was originally a technique to reduce parallel time, in the sense that $m$-bridging reduces communication on distributed-memory machines, it is also a technique to improve spatial locality. We have focused particular attention on $m$-bridging as a package of locality enhancement and parallelization, and then applied it to iterative tree traversal.

In this chapter, we present an application of $m$-bridging to locality-aware parallelization of iterative tree traversal. The main idea of our approach is recursive application of $m$-bridging and recursive storing of querying points. Owing to the genericity of $m$-bridging, our approach does not necessitate balancing of space-partitioning trees or even require any application-specific knowledge, yet provides cache efficiency and load balancing. This virtue of our approach is advantageous to a generic implementation of iterative tree traversal.

The following are our contributions:

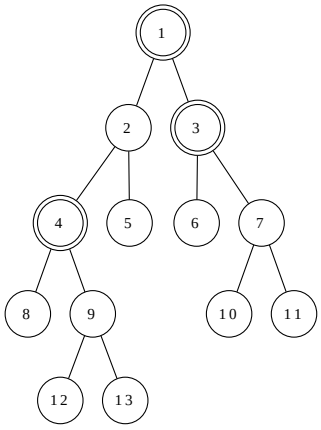



Figure 12.1: A selection of hole nodes, where a double-circle node denotes a hole node.

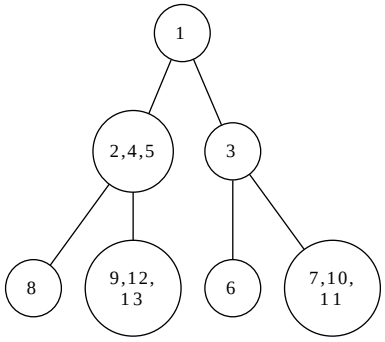



Figure 12.2: The coarser tree derived from the selection of hole nodes in Figure 12.1.

- We formulate iterative tree traversal as a parallel pattern iter (Section 12.3).
- We have designed data structures for iterative tree traversal on the basis of $m$-bridging [Rei93], analyzing cache complexity (Section 12.4).
- We describe the applications of iter and our data structures in several problems (Section 12.5).

## 12.2 Preliminaries

We introduce several notions on trees. The tree in this section means a full binary tree.

### 12.2.1 Segmentation of Trees

In the implementation of tree skeletons [Mat07a, SM14a] as explained in Chapters 2, segmented trees are used as input trees, For any selection of hole nodes, we can define a segmentation, i.e., a transformation from a simple tree to a segmented tree. This segmentation therefore can be seen as an operation of coalescing nodes of a given tree. For example, from a selection of hole nodes as illustrated in Figure 12.1, we obtain a *coarser tree*, whose nodes are segments, as illustrated in Figure 12.2. In this chapter, we consider the construction of segmented trees as a node-coalescing operation.

### 12.2.2 $m$-Bridge

Given a tree, how to select hole nodes is an issue on segmentation. As a graph-theoretic result, the $m$-critical criterion is known to be convenient.

**Definition 1** ($m$-critical node [Rei93])**.** Let $m$ be an integer such that $1 < m \leqslant N$ and $N$ is the number of nodes in a tree. A node $v$ in the tree is called $m$-critical if $v$ is an internal node and $\lceil weight(v)/m \rceil > \lceil weight(v_c)/m \rceil$ for each child $v_c$ of $v$, where $weight(u)$ denotes the number of the nodes of the subtree rooted by a node $u$.

Each segment divided by $m$-critical nodes is called an *$m$-bridge*. We call it *$m$-bridging* to coalesce nodes on the basis of $m$-critical nodes. For example, the hole nodes illustrated in Figure 12.1 are 4-critical nodes and the coarser tree illustrated in Figure 12.1 is the result of 4-bridging.

For $m$-bridges of trees, the following two important properties are known.

**Lemma 1** ([Rei93])**.** *The number*[1] *of the nodes of an $m$-bridge is at most $m$.*

**Lemma 2** ([Rei93])**.** *Let $N$ be the number of the nodes of a tree. The number of $m$-critical nodes of the tree is at most $2N/m - 1$.*

Since the number of the segments of a segmented tree that contains $n$ hole nodes is $2n + 1$, we can rephrase Lemma 2 as follows.

**Lemma 3** (Rephrasing of Lemma 2)**.** *Let $N$ be the number of the nodes of a tree. The number of $m$-bridges of the tree is at most $4N/m - 1$.*

These properties mean that $m$-bridging is a decomposition from a tree to sufficiently balanced segments. This contributes significantly to asymptotic linear speedup of operations on segmented trees.

Another important point of $m$-bridging is its cheapness. By using the upward accumulation skeleton (i.e., uAcc [Ski96, GCS94]), we can mark $m$-critical nodes. Although constructing and restructuring data structures will take some cost in practice, $m$-bridging itself is a cheap computation and will not be a sequential bottleneck.

## 12.3 Iterative Tree Traversal

In this section, we formulate iterative tree traversal as a parallel pattern (i.e., skeleton [RG02]). We first define a space-partitioning tree as

$$SPTree_{\alpha,\beta} = Branch(x, SPTree_{\alpha,\beta}, SPTree_{\alpha,\beta}),$$
$$SPTree_{\alpha,\beta} = Leaf(y),$$

where $x$ of type $\alpha$ denotes a subspace, which may contain its splitting criteria, and $y$ of type $\beta$ denotes a point (or a minimum unit of space).

The iterative tree traversal that we suppose takes a space-partitioning tree $\mathcal{T}$ of type $SPTree_{\alpha,\beta}$ and a set $Q$ of querying points (or units of querying), and yields a set that has one-to-one correspondence to $Q$. Although the order in $Q$ is unnecessary, a list is convenient to discriminate the elements of $Q$ and to formulate operations between $Q$ and a resultant set. We therefore use a list of type *List* for both $Q$ and a resultant set, where $List_a$ denotes a list whose elements are type of $a$. Letting $Q$ be type of $List_\gamma$, we generally suppose that $\gamma$ may differ from $\beta$ of $SPTree_{\alpha,\beta}$. For example, in range queries, $\gamma$ will denote a range and $\beta$ may denote a point.

[1] It is at most $m + 1$ in [Rei93] because every segment there except for root segments is rooted by a replicated hole node of its parent segment.

We define iterative tree traversal as the following higher-order function iter:

$$
\begin{aligned}
&\mathsf{iter}: \ (\gamma \times \alpha \to \mathrm{bool}) \times (\gamma \times \alpha \to \delta) \times (\gamma \times \beta \to \delta) \\
&\qquad \times (\delta \times \delta \to \delta) \times \mathit{SPTree}_{\alpha,\beta} \times \mathit{List}_{\gamma} \to \mathit{List}_{\delta} \\
&\mathsf{iter}(c, k_b, k_l, \oplus, \mathcal{T}, Q) = \mathsf{map}(f, Q), \\
&\text{where } f(x) = h(x, \mathcal{T}), \\
&\qquad h(q, \mathit{Branch}(x, t_L, t_R)) = \textbf{if } c(q, x) \textbf{ then } k_b(q, x) \textbf{ else } h(q, t_L) \oplus h(q, t_R), \\
&\qquad h(q, \mathit{Leaf}(y)) = k_l(q, y), \\
&\qquad \{\text{Algebraic condition}\} \\
&\qquad \oplus \text{ is associative and commutative.}
\end{aligned}
$$

Here map is the standard one of list functions; it takes a unary function and a list, and then yields a list by applying the function to each element of the list. Note that map has embarrassing parallelism. The function $h$ traversing $\mathcal{T}$ consists of the top-down pruning with a criterion $c$ and the bottom-up reduction with operators $k_b$, $k_l$, and $\oplus$. The associativity and commutativity of $\oplus$ bring a high degree of parallelism. Note that both stem semantically from a degree of freedom in space partitioning rather than artificial conditions.

The iter above does not capture all kinds of tree traversals but does hit a sweet spot as a parallel pattern. The discussions on its applicability shall be described in Section 12.5. iter is an example model of iterative tree. Although iter may has a high degree of parallelism regarding map, each tree traversal is sequential. To obtain a high degree of parallelism from a tree traversal, we have to define the tree traversal of iter for segmented trees. Actually, we can define it naturally without any additional operator.

We first formulate segmented trees of $\mathit{SPTree}_{\alpha,\beta}$ as the following $\mathit{SegTree}_{\alpha,\beta}$:

$$
\begin{aligned}
\mathit{SegTree}_{\alpha,\beta} &= \mathit{Branch}(x, x_\bullet, \mathit{Ctx}_{\alpha,\beta}, \mathit{SegTree}_{\alpha,\beta}, \mathit{SegTree}_{\alpha,\beta}), \\
\mathit{SegTree}_{\alpha,\beta} &= \mathit{Sub}_{\alpha,\beta}, \\
\mathit{Ctx}_{\alpha,\beta} &= \mathit{Branch}(x, \mathit{Ctx}_{\alpha,\beta}, \mathit{Ctx}_{\alpha,\beta}), \\
\mathit{Ctx}_{\alpha,\beta} &= \mathit{Hole}, \\
\mathit{Ctx}_{\alpha,\beta} &= \mathit{Sub}_{\alpha,\beta}, \\
\mathit{Sub}_{\alpha,\beta} &= \mathit{Sub}(\mathit{SPTree}_{\alpha,\beta}),
\end{aligned}
$$

where $\mathit{Ctx}_{\alpha,\beta}$ denotes a one-hole context, $\mathit{Sub}_{\alpha,\beta}$ denotes a subtree, and $x_\bullet$ denotes the subspace of a hole node. Since hole nodes are internal nodes, $x_\bullet$ is type of $\alpha$. *Hole* is nullary because $x_\bullet$ is placed at the next of the root of a one-hole context. The grammar above per se does not guarantee that $\mathit{Ctx}_{\alpha,\beta}$ contains one *Hole*. From the definition of segmented trees, the following is an invariant:

$$x_1 = x_2 \text{ for } \mathit{Branch}(x_1, x_\bullet, \mathit{Branch}(x_2, s_1, s_2), t_1, t_2).$$

We then define a traversal function $h_s$ for $\mathit{SegTree}_{\alpha,\beta}$ that corresponds to $h$ in the where clause of iter as follows:

$$
\begin{aligned}
&\{c,\ k_b,\ k_l, \text{ and } \oplus \text{ are given as } h \text{ of } \mathsf{iter}\} \\
&h_s: \ \gamma \times \mathit{SegTree}_{\alpha,\beta} \to \delta \\
&h_s(q, \mathit{Branch}(x, x_\bullet, s, t_L, t_R)) = \textbf{if } c(q, x) \textbf{ then } k_b(x) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else if } c(q, x_\bullet) \textbf{ then } h_c(q, k_b(x_\bullet), s) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else } h_c(q, \iota_\oplus, s) \oplus h_s(q, t_L) \oplus h_s(q, t_R), \\
&h_s(q, \mathit{Sub}(t)) = h(q, t), \\
&h_c(q, d_\bullet, \mathit{Branch}(x, t_L, t_R)) = \textbf{if } c(q, x) \textbf{ then } k_b(q, x) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else } h_c(q, d_\bullet, t_L) \oplus h_c(q, d_\bullet, t_R), \\
&h_c(q, d_\bullet, \mathit{Hole}) = d_\bullet, \\
&h_c(q, d_\bullet, \mathit{Sub}(t)) = h(q, t),
\end{aligned}
$$

where $\iota_\oplus$ denotes the identity of $\oplus$.

## 12.4 Proposed Data Structures

In this section, we describe our cache-efficient data structures for iterative tree traversal.

We adopt the cache-oblivious model [FLPR99] for analyzing cache complexity. A cache consists of $Z$ words and a cache line (i.e., a unit of cache replacement) consists of $L$ words, where $L^2 < Z$ (i.e., tall cache) and ideal replacement are assumed. We assume input size $N$ to be much larger than $Z$, i.e., $Z \ll N$. We assume that each node containing some values and querying points consists of constant-size continuous words.

### 12.4.1 Simply Blocked Tree

We first describe a simple approach to reducing the cache complexity of tree traversal as a counterpart of proposed data structures.

#### Overview

To reduce the cache misses, we generally have to use arrays for obtaining continuous memory access. We here consider blocking of a given full binary tree of $N$ nodes. Let $B_t$ be an integer such that $1 < B_t \leqslant N$. A block in a given tree $\mathcal{T}$ is a tree which is rooted by the root of $\mathcal{T}$ or a child of a hole node, and whose leaves are leaves of $\mathcal{T}$ or hole nodes. Similar to $m$-bridging, we can mark nodes as holes by calculating their weights in a bottom-up manner. An internal node $v$ is a hole if $\lceil \mathit{weight}(v)/B_t \rceil > \lceil \mathit{weight}(v_c)/B_t \rceil$ holds for $v$ and each child $v_c$ of $v$, where $\mathit{weight}(v)$ denotes the number of the nodes in a block rooted by $v$. We call a tree whose nodes in the same block are successively arranged in an array a *simply blocked tree* and call this blocking of trees *$B_t$-blocking*.

#### Analysis

We first consider the usual behavior of a single query of iter. The worst-case behavior is immediate; the whole tree is traversed if a pruning function $c$ always returns false. This behavior, however, does not take account of the usage of space-partitioning trees. Given a querying point, we usually use space-partitioning trees for finding its neighbors and pruning faraway ones. In this case, the trace of a query tends to form a kind of *spine*: a root-to-leaf path like a stem with short branching arms. We consider such spine-like traces as usual behaviors. If we assume that the branching arms of a spine-like trace is constant-length, it can approximate to the root-to-leaf path of its stem. It is therefore reasonable to analyze the traversal of a root-to-leaf path as an approximation.

The $B_t$-blocking of full binary trees has an effect similar to $m$-bridging. Each block has at most $B_t$ nodes. Let $N$ be the number of the nodes of a tree to which blocking is applied. The number of the hole nodes is at most $\lceil N/B_t \rceil - 1$ since the total number of the nodes in the two blocks whose parent is a hole node is at least $B_t$. The number of the blocks is at most $2\lceil N/B_t \rceil - 1$ since each hole node generate two blocks except for the root block.

On the basis of the properties above of simply blocked trees, we analyze the cache complexity of the traversal of a root-to-leaf path. Letting $N_B$ be the number of blocks, a root-to-leaf path overlaps at most $N_B/2 + 1$ blocks. The cache misses caused by the traversal of each block is bounded by $\mathrm{O}(\lceil B_t/L \rceil)$. The cache complexity of the traversal of a root-to-leaf path is therefore $\mathrm{O}(\lceil B_t/L \rceil \lceil N/B_t \rceil)$, which reduces to $\mathrm{O}(\lceil N/L \rceil)$ if $B_t \geqslant L$.

**Theorem 1.** *Let $\mathcal{T}$ be a simply blocked tree derived from $B_t$-blocking of a given full binary tree of $N$ nodes. If $B_t \geqslant L$, the cache complexity of the traversal of a root-to-leaf path on $\mathcal{T}$ is* $\mathrm{O}(\lceil N/L \rceil)$.

If we use a naive linked-structure tree, the cache complexity of the traversal of a root-to-leaf path is $\mathrm{O}(N)$ because the given tree might have $\mathrm{O}(N)$ height, i.e., be a list-like tree. In the case of a balanced binary tree, naive linked-structure trees cost $\mathrm{O}(\lg N)$ cache misses. Meanwhile, simply blocked trees cost

$O(\lceil B_t/L \rceil \log_{B_t} N)$ cache misses since a root-to-leaf path overlaps at most $\log_{B_t} N$ blocks. If $B_t = O(L)$, it reduces to $O(\log_L N)$ cache misses.

#### Pros and Cons

As a general technique to improve cache complexity, $B_t$-blocking has two major advantages. First, $B_t$-blocking does not sacrifice applications of space-partitioning trees. For example, it does not necessitate any change in iter and enables us to use a simple definition of iter for $SPTree_{\alpha,\beta}$ unlike segmented trees. This is not limited to $SPTree_{\alpha,\beta}$ and iter. Second, $B_t$-blocking is as cheap as $m$-bridging. We do not need a careful treatment of its overhead.

$B_t$-blocking has a drawback related to the height of resultant trees. Although space-partitioning trees do not become extremely imbalanced (i.e., list-like) in practice, the effect of $B_t$-blocking becomes limited, relying on the shape of a given tree. In addition, the height of the tree of blocks corresponds directly to the depth of parallel complexity. A larger height leads to a worse depth.

### 12.4.2 Recursively Segmented Tree

The main difference between $B_t$-blocking and $m$-bridging is that a block may have multiple hole nodes, while a bridge has at most one hole node. On the basis of this property, we can improve the cost of tree traversal in some cases. We next describe the usage of $m$-bridging for improving the cache complexity of space-partitioning trees.

#### Overview

$m$-bridging introduces a hierarchy of a tree. A coarser tree is smaller than its original tree but each node of a coarser tree is larger than each node of its original tree. If both a coarser tree and each of its segments fit in cache, cache misses in tree traversal will be much reduced.

Our main idea is to apply $m$-bridging recursively to a given space-partitioning tree. Note that $weight(x)$ used in the $m$-critical criterion for a coarser tree does not count the number of the nodes in a segment; that is, $weight(x)$ regards a coarser tree as a simple tree. We consider a level of segments. We call the coarsest segment that includes the whole the *root-level* (or 0th-level) segment and call the segments of an original tree *last-level* segments. We name such a hierarchical tree a *recursively segmented tree* derived from $m$-bridging. We consider that the nodes of each tree on any level are stored continuously into an array. This design improves the spacial locality of tree traversal.

An important concern is whether iter is definable on recursively segmented trees. This is immediate. In the case of recursively segmented trees, the definition of $Sub_{\alpha,\beta}$ is extended by the following rule:

$$Sub_{\alpha,\beta} = Seg(SegTree_{\alpha,\beta}).$$

Then, the definitions of $h_s$ and $h_c$ are extended by the following cases:

$$h_s(q, Seg(t)) = h_s(t),$$
$$h_c(q, d_\bullet, Seg(t)) = h_s(t).$$

Another important concern is how to accelerate traversals. In traversing a root-to-leaf path, the definition of iter for $SPTree_{\alpha,\beta}$ traverse all its nodes. If all these are necessary for calculating the result of a query, it is difficult to obtain better cost than simply blocked trees. However, we can sometimes ignore internal nodes in practice. For example, in the cases of range queries (Section 12.5.1) and a $k$-nearest-neighbor query (Section 12.5.4), we are concerned only with leaf values. In such cases, we can skip internal nodes and shortcut the traversal to a leaf. To enable such shortcut acceleration, we introduce two additional parameters to iter. One is a skipping criterion $c_c$ that takes a querying point and a one-hole subspace, and yields whether we can skip the target one-hole subspace. The other is a function $k_c$ that calculates the contribution from one-hole subspace to a querying point. For example,

if we can ignore internal nodes completely, $k_c$ is a constant function yielding $\iota_\oplus$. Then, we extend the definition of $h_s$ as follows:

$$
\begin{aligned}
&\{s \text{ and } k_c \text{ as well as } c,\ k_b, \text{ and } \oplus \text{ are given}\}\\
&h_s(q, \mathit{Branch}(x, x_\bullet, s, t_L, t_R)) = \textbf{if } c(q, x) \textbf{ then } k_b(x)\\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else if } c(q, x_\bullet) \textbf{ then } h_c(q, k_b(x_\bullet), s)\\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else if } c_c(q, x, x_\bullet) \textbf{ then } k_c(q, x, x_\bullet) \oplus h_s(q, t_L) \oplus h_s(q, t_R)\\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else } h_c(q, \iota_\oplus, s) \oplus h_s(q, t_L) \oplus h_s(q, t_R).
\end{aligned}
$$

If $c_c$ returns true, $h_c$ prunes the traversal of the next-level segment and instead applies $k_c$ to the one-hole subspace, i.e., skips an internal segment.

#### Analysis

To determine $m$-bridging appropriate to recursively segmented trees, we analyze the cache complexity for a simple query finding a leaf, which we call a root-to-leaf traversal. In this case, we can assume $c_c$ always returns true in iter. For simplicity, we assume $m > 4$.

From Lemma 3, we can assume that the number of the levels of a recursively segmented tree derived from $m$-bridging is at most $\log_{m/4}(N/m)$, where $N$ is the number of the nodes of its original tree. Since from Lemma 1 any segment on any level fits into $\mathrm{O}(m)$ words, the cache misses caused by the traversal of a segment is bounded by $\mathrm{O}(\lceil m/L \rceil)$. To find a leaf from the root, we traverse $\mathrm{O}(\log_{m/4}(N/m))$ segments. The total cache misses in a root-to-leaf traversal are bounded by $\mathrm{O}(\lceil m/L \rceil \log_{m/4}(N/m))$, which reduces to $\mathrm{O}(\log_L N)$ if $m = \mathrm{O}(L)$, e.g., $4L$.

**Theorem 2.** *Let $\mathcal{T}$ be a recursively segmented tree derived from $m$-bridging of a full binary tree that has $N$ nodes. If $m = \mathrm{O}(L)$, the cache complexity of a root-to-leaf traversal on $\mathcal{T}$ is $\mathrm{O}(\log_L N)$.*

#### Pros and Cons

The skip of internal segments imposes a restriction on space-partitioning trees in practice. For example, vantage-point (VP) trees require a top-down one-by-one traversal for an effective pruning because the pruning criterion for a subspace uses its ancestors' subspaces implicitly. Pruning without ancestors' subspaces is safe but would be ineffective. Therefore, recursively segmented trees derived from VP trees are ineffective. To use recursively segmented trees in practice, we have to consider this additional restriction.

The definitions of $c_c$ and $k_c$ are not always straightforward. We explain several examples in Section 12.5. If we are concerned only with leaf values, the definitions of $c_c$ and $k_c$ are straightforward. All in all, to define reasonably $c_c$ and $k_c$ necessitates domain knowledge to some extent.

Although we have to deal with an additional restriction and requirement to use recursively segmented trees, they can guarantee that a root-to-leaf traversal causes $\mathrm{O}(\log_L N)$ cache misses regardless of the shape of a given tree. That is, recursively segmented trees are more restrictive and efficient than simply blocked trees. In this sense, both have complementary characteristics.

### 12.4.3 Buffered Recursively Segmented Tree

Our aim is to capture an iterative nature in iterative tree traversal. We therefore sophisticate recursively segmented trees for iterative querying.

#### Overview

A problematic case in iterative querying is that the trace of one query is quite different from that of the next query. Such successive queries have a poor temporal locality. For a good temporal locality, successive queries have to cause similar traces. When a series of different queries is given, we should therefore reorder these queries for improving temporal locality.

Our method of reordering queries for recursively segmented trees is to buffer querying points at each segment. In a appropriately small segment, all queries tend to be similar because possible traces are not many. All queries that reach a next-level segment also tend to be similar because quite different queries do not reach it. To improve temporal locality, it is therefore sufficient to buffer querying points at each segment on each level and postpone their traversals. Specifically, we equip each segment with a buffer[2] of size $B$ to postpone traversals on it. We assume that the buffers of the sibling segments on the same level are a continuous array. When a buffer becomes full, buffered querying points traverse the associated segment and are stored into the buffers of the next-level segments. We name such a buffered version a *buffered recursively segmented tree*. Note that the buffered version of simply blocked trees, *a buffered simply blocked tree* is also feasible.

### Analysis

We analyze the cache complexity of iterative root-to-leaf traversal of a buffered recursively segmented tree $\mathcal{T}$. Let $Q$ be a set of querying points. We assume that each buffer size $B$ is the size of $L$ querying points and $m = \mathrm{O}(L)$. We first have to consider a worst-case series of queries regarding cache complexity. That one path is completely different from the next path is that the intersection of the segments of both traces is only the root-level segment. A series of such trace paths causes uniformly fill each buffer on the same level. That is, first $L$ querying points fill the buffer of the root-level segment; first $L + L^2$ querying points fill all the buffers of the root-level and first-level segments; first $L + L^2 + L^3$ queries fill all the buffers of the root-level, first-level, and second-level segments; ...; this is a worst case. Since $L^2 < Z$ and sibling buffers are continuous, flushing the full buffer of a segment into the next-level segments without flushing their buffers causes $\mathrm{O}(L)$ cache misses. The flushes of the $i$th-level buffers cause $\left\lceil |Q|L^{-i-1} \right\rceil$ and a flush of the $i$th-level buffers cause $\left\lceil L^i \right\rceil$ cache misses. The total cache misses of $i$th-level buffers are bounded by $\mathrm{O}(\lceil |Q|/L \rceil)$. Since the number of levels is bounded by $\mathrm{O}(\log_L N)$, the total cache misses of the iterative root-to-leaf traversal of $\mathcal{T}$ regarding $Q$ are bounded by $\mathrm{O}(\lceil |Q|/L \rceil \log_L N)$.

**Theorem 3.** *Let $\mathcal{T}$ be a buffered recursively segmented tree derived from $m$-bridging and equipped with buffers of size $B$ and $N$ be the number of the nodes of its original tree. The cache complexity of iterative root-to-leaf traversal of $\mathcal{T}$ regarding a set $Q$ of querying points is $\mathrm{O}(\lceil |Q|/L \rceil \log_L N)$ if $m = \mathrm{O}(L)$ and $B = \mathrm{O}(L)$.*

We can easily analyze the cache complexity of iterative traversal of a root-to-leaf path on a buffered simply blocked tree $\mathcal{T}'$. We assume $B_t = \mathrm{O}(L)$. Letting $N_B$ be the number of blocks, a root-to-leaf path overlaps at most $N_B/2 + 1$ blocks. The worst case is the iterative traversal of the longest block path. By buffering querying points, the all buffers in the path flush per $L$ queries in the worst case and the cache misses per flush of a buffer are bounded by $\mathrm{O}(1)$. the total cache misses are $\mathrm{O}(\lceil |Q|/L \rceil \lceil N/L \rceil)$.

**Theorem 4.** *Let $\mathcal{T}$ be a buffered simply blocked tree derived from $B_t$-blocking with buffers of size $B$ and $N$ be the number of the nodes of its original tree. The cache complexity of the iterative traversal of a root-to-leaf path of $\mathcal{T}$ regarding a set $Q$ of querying points is $\mathrm{O}(\lceil |Q|/L \rceil \lceil N/L \rceil)$ if $B_t = \mathrm{O}(L)$ and $B = \mathrm{O}(L)$.*

If we use a naive linked-structure tree, the cache complexity of the traversal of a root-to-leaf path is $\mathrm{O}(|Q|N)$. In the case of a balanced binary tree, naive linked-structure trees cost $\mathrm{O}(|Q| \lg N)$ cache misses. Meanwhile, letting $B_t = \mathrm{O}(L)$, buffered simply blocked trees cost $\mathrm{O}(\lceil |Q|/L \rceil \log_L N)$ cache misses through an analysis similar to one for buffered recursively segmented trees.

### Handling Accumulation

Unfortunately, the asymptotic cost above does not model the cost of iter sufficiently. iter performs the reduction with $\oplus$ of values that are generated at the leaves of a trace. It raises an issue on cache misses. Even though a root-to-leaf traversal can be an approximation of the trace of a query as mentioned earlier, a trace has multiple leaves. To reduce them, an accumulation for each querying point is necessary. A

[2] Although a buffer is useless for the root-level segment, we introduce it for brevity of explanation.

matter is the buffer for this accumulation. A common yet space-efficient way is to use a resultant array of iter for a accumulating buffer. Since replications of querying points are stored in buffers and their traversals are postponed, accumulations on each element of a resultant array will be, however, noncontinuous. In the worst case, a cache miss results at each accumulation of a value.

In addition to a resultant array, we can introduce accumulation buffers into each segment. Similarly to the top-down pruning phase, the bottom-up reduction phase can be buffered and postponed. This approach reduces noncontinuous accumulations to a resultant array. Still, it is difficult to improve the worst-case cache complexity. Moreover, this approach increases constant factors in space complexity and complicates implementation.

A reasonable heuristic is to split $Q$ into continuous blocks, each of which is denoted by $Q_B$, and force $\mathcal{T}$ to finish all postponed queries per $Q_B$. If $|Q_B| = \mathrm{O}(Z)$, the segment of a resultant array corresponding to $Q_B$ fits in the cache. Without tree traversals, the cache misses caused by the accumulations regarding $Q_B$ would be bounded by $\mathrm{O}(\lceil Q_B/L \rceil)$. Because the tree traversals for $Q_B$ may flush the whole cache, it cannot be the worst-case bound of the reduction part of iter. However, by changing the block size under $L < |Q_B| < Z$, we can control the probability of cache misses. This heuristic with tuning on $|Q_B|$ is a practically reasonable choice for reducing the cache misses in iter.

### 12.4.4 Parallel Implementation

Although we have focused only on locality and cache complexity, our aim is to package locality enhancement and parallelization. We now describe how to parallelize iter on buffered recursively segmented trees. We suppose uniform distributed caches. Let $P$ be the number of processors.

#### Simple Parallelization

iter has embarrassing parallelism regarding a given set $Q$ of querying points. It is reasonable to divide $Q$ evenly to $P$ processors. Although the cost of each query may be different, randomization can resolve such imbalance in practice. Since blocking querying points is valuable for locality, randomization should be performed per $Q_B$.

Owing to the associativity and commutativity of $\oplus$, the reduction part can be parallelized. Because a set of values for reduction is usually much smaller than $|Q|$, it is not worth parallelizing the reduction for a querying point with an additional synchronization cost.

Synchronization should be least. The parallelization of iter regarding $Q$ for $SPTree_{\alpha,\beta}$ requires only a barrier synchronization at the end. For buffered recursively segmented trees, the race on the buffer of each segment results. The mutual exclusion for each access to buffers is expensive. To elude race on buffers, it is, however, sufficient to replicate the buffers of all segments for each processor. This privatization of buffers prevents false sharing on a hierarchical cache in common multicore processors and therefore is practically valuable.

#### Distributed Parallelization

Although the simple parallelization above utilizes only parallelism regarding $Q$, we can utilize parallelism on recursively segmented trees by distributing segments into each processor. Since this distribution reduces the amount of shared data among all processors, this distributed parallelization is beneficial to implementation on distributed-cache/-memory machines.

For load balancing of tree skeletons on $m$-bridged trees, the case of $m = 2N/P$ guarantees asymptotic linear speedup with maximum spatial locality. The case of smaller $m$, however, brings better load balancing by sacrificing spatial locality. An appropriate range of $m$ was explored in [Mat07b]. These are based on single $m$-bridging and therefore schedule only last-level segments. Even though we can schedule only last-level segments for recursively segmented trees, such too fine-grained scheduling causes a poor spatial locality. We therefore have to consider to coarse-grained scheduling for recursively segmented trees.

An important point is that forming recursively segmented trees and load balancing demand different criteria on $m$-bridging. Specifically for load balancing, letting $s$ be a segment, $\mathit{weight}(s)$ in the $m$-critical

criterion must be the number of the nodes of its original tree. We therefore have to apply an additional $m$-bridging to recursively segmented trees for load balancing. We use $m$ for the parameter of forming recursively segmented trees and $m'$ for that of load balancing, assuming $m < m'$.

The calculation of $m'$-critical nodes on recursively segmented trees is straightforward. A problem is that an $m'$-bridge does not necessarily match any segment of a recursively segmented tree. Even if an $m'$-bridge does not match a segment, it can contain some of its finer-level segments. With a space-containment criterion $c' : \alpha \times \alpha \to \text{bool}$ for $SPTree_{\alpha,\beta}$, we can assign processors to segments that cover all nodes of an original tree.

Although processors are not assigned to coarse-level segments (especially, root-level ones), it is no problem, rather convenient. Many querying points will pass through such coarse-level segments in traversing different $m'$-bridges. It is therefore reasonable to share or replicate them among all processors. In addition, $m'$-bridges of much smaller weight such as single-node ones (see Figure 12.2) are not worth privatizing in practice. We therefore may neglect assigning processors to segments contained in such cheap $m'$-bridges.

When an $m'$-bridge contains a segment, we do not have to assign a processor explicitly to the finer-level segments contained in it. That is, we can prune processor assignment in top-down traversal. The computational pattern of this processor assignment is therefore very close to iter. Intuitively, a containment criterion $c'$ corresponds to a pruning criterion $c$ in iter, and the destructive updating of nodes regarding processor numbers corresponds to $k_b$ and $k_l$ in iter. Note that the assignment of processor numbers does not cause race by definition.

By using the method above, we can distribute buffered recursively segmented trees. It is, however, insufficient for tree traversal. We have to consider intersection (or communication) among processors in tree traversal. We can implement it as a simple asynchronous parallel computing. We associate a mailbox, which is an asynchronous buffer, with each segment numbered a processor number. When one processor transmits a querying point to a segment with a mailbox for another processor, we insert the querying point into the mailbox. Each processor always monitor its own mailboxes and transmit querying points in an own mailbox to the buffer of segment with which the mailbox is associated. By using such an asynchronous intersection, we can implement tree traversal on distributed buffered recursively segmented trees.

In summary, how to distribute a buffered recursively segmented tree $\mathcal{T}$ is the following:

1. Calculate each $m'$-critical node on $\mathcal{T}$ and obtain its subspace, which may have a hole.
2. Distribute $m'$-critical subspaces to all processors, where subspaces of much smaller weight may be neglected.
3. Assign processor numbers to segments by using a space-containment criterion $c'$ in a manner similar to iter.
4. Associate a mailbox for processor $i$ with each segment numbered $i$.

## 12.5 Applications

In this section, we describe applications of iter and explore the potential of our data structures on these applications.

### 12.5.1 Batch Processing with Range Queries

In application of databases, batch processing is important. This usually performs a routine analysis on data that is growing. Since the focus of such cases is on the difference from the result of the previous analyses, an analysis query often contains a range of time such as last 24 hours. In addition to time, we may consider other continuous data, e.g., prices and spatial position. A typical analysis query summarizes the data of concerned ranges regarding a concerned key. If a set of concerned keys is given, its batch processing will iterate such summarizing queries. iter covers such batch processing with range queries.

For example, we consider a database of trade history. A trade is a triple of type $Id \times Time \times Price$, where $Id$ denotes trader's identity numbers, $Time$ denotes trade timestamps, and $Price$ denotes trade prices. The space partitioned by a given tree $\mathcal{T}$ is two-dimensional of type $Id \times Time$ and its subspaces are rectangles of type $Rect_{Id \times Time}$. A leaf subspace of $\mathcal{T}$ is a trade list of type $List_{Id \times Time \times Price}$. A querying point is a line segment on $Time$ of type $Id \times Period$, where $Period$ denotes the type of periods, and a query sums up the trade prices on this line segment. That is, this batch processing calculates the sales of traders during some period. The types and parameters of iter for this batch processing are defined as follows.

$$
\begin{aligned}
\alpha &= Rect_{Id \times Time}, \\
\beta &= List_{Id \times Time \times Price}, \\
\gamma &= Id \times Period, \\
\delta &= Price, \\
c(q, x) &= isEmpty(q \cap x), \\
k_b(q, x) &= 0, \\
k_l((i, p), y) &= \sum [w \mid (j, t, w) \in y,\ i = j,\ t \in p], \\
x \oplus y &= x + y, \\
c_c(q, x, x_\bullet) &= q \subseteq x_\bullet, \\
k_c(q, x, x_\bullet) &= 0,
\end{aligned}
$$

where $[\ldots \mid \ldots]$ is a notation for list comprehension and $\sum [\ldots \mid \ldots]$ denotes list reduction with $+$.

If we perform distributed parallelization, we obtain a distributed database immediately. Since the insertion of trades to this distributed $\mathcal{T}$ does not necessitate mutual exclusion except for mailbox, it is therefore suited to data updated every day. Although balanced $k$-d trees are known to be appropriate for range queries, the advantage of our recursively segmented trees is that its (re)construction does not have to take the definition of space into account.

### 12.5.2 N-body Problem

The $N$-body problem is to calculate the contributions among $N$ particles. Although the direct method costs $\mathrm{O}(N^2)$ time, the Barnes-Hut algorithm, which utilizes some approximation, costs $\mathrm{O}(N \log N)$ time. It typically uses an octree to hold $N$ particles in the three-dimensional space. The main idea of the Barnes-Hut algorithm is to approximate faraway particles by using a subspace containing them. This is a pattern of iter where a space-partitioning tree contain all querying points.

For example, in gravitation simulations, a query calculates a force of type $Force$, which is a three-dimensional vector. Particles (i.e., mass points) are type of $Particle$, which has to hold a pair of mass and position for gravitation. The subspaces of the three-dimensional space are type of $Rect_3$. Each node of a given octree has a pair of its subspace and a particle that approximates the particles contained in its subspace. In addition to this pair, each leaf subspace has a particle list of type $List_{Particle}$. The types

and parameters of iter for the Barnes-Hut algorithm are defined as follows.

$$
\begin{aligned}
\alpha &= \mathit{Rect}_3 \times \mathit{Particle},\\
\beta &= \mathit{Rect}_3 \times \mathit{Particle} \times \mathit{List}_{\mathit{Particle}},\\
\gamma &= \mathit{Particle},\\
\delta &= \mathit{Force},\\
c(q,x) &= \mathit{isFaraway}(q,x),\\
k_b(q,(r,p)) &= \mathit{force}(p,q),\\
k_l(q,(r,p,L)) &= \textbf{if } c(q,(r,p)) \textbf{ then } \mathit{force}(p,q) \textbf{ else } \sum[\mathit{force}(p,q) \mid p \in L,\ p \neq q]\\
x \oplus y &= x + y,\\
c_c(q,x,x_\bullet) &= \mathit{isFaraway}(q, x - x_\bullet),\\
k_c(q,x,x_\bullet) &= \mathit{force}(q,p),\\
&\quad \text{where } (r,p) = x - x_\bullet,
\end{aligned}
$$

where $\mathit{isFaraway}(q,x)$ denotes a faraway criterion of particle $q$ from subspace $x$, $\mathit{force}(p,q)$ denotes a force from particle $p$ to particle $q$, and $+$ denotes a vector addition. Although $\mathit{SPTree}_{\alpha,\beta}$ is a binary tree, it can represent any octree naturally by dividing each dimension successively.

In the definitions above of $c_c$ and $k_c$, we construct a one-hole subspace through the subtraction over $\mathit{Rect}_3 \times \mathit{Particle}$. Although the subtraction over $\mathit{Rect}_3$ is immediate, the subtraction over *Particle* requires a consideration. Recall that the part of *Particle* denotes an approximation of the points contained in a subspace. In the Barnes-Hut algorithm, The total mass and the center of mass are usually used for this approximation. In this case, we can define the addition of *Particle* as the calculation of total mass and that of the center of mass, and then define the subtraction as its cancellation operation. In addition, we have to define $\mathit{isFaraway}(q,x)$ for not only $x$ of type $\mathit{Rect}_3$ but the one-hole subspaces. Although a safe definition of *isFaraway* is easy, a reasonable definition is important for effective pruning.

### 12.5.3 Ray Tracing

Ray tracing is a popular problem in computer graphics. It traces lines of sight (i.e., view rays) from a viewpoint to objects settled in the three-dimensional space. An intersection point of view rays to objects leads to a pixel of a resultant image. A given set of objects settled in the three-dimensional space is called a *scene*.

In ray tracing algorithms, a bounding volume hierarchy (BVH) is often used for detecting intersections in a scene. The BVH decomposes a bounding box into finer ones and holds an object at a leaf. It enables us to prune intersection tests.

Photon mapping is the most popular algorithm for ray tracing. It scatters many photons over a scene in advance and stores them usually into a $k$-d tree. Then, it collects photons around an intersection point from the $k$-d tree and estimates the illumination at the point.

We can apply iter to both cases. In both cases, subspaces are type of $\mathit{Rect}_3$.

Each leaf of a BVH holds an objects of type *Object*, which is a supertype of $\mathit{Rect}_3$. A querying point is a view ray of type *Ray*. A query calculates an intersection position of type *Position*. In the case of

BVHs, the types and parameters of iter are defined as follows.

$$\begin{aligned}
\alpha &= \mathit{Rect}_3, \\
\beta &= \mathit{Object}, \\
\gamma &= \mathit{Ray}, \\
\delta &= \mathit{Position}, \\
c(q, x) &= (\mathit{intersect}(q, x) \equiv p_\infty), \\
k_b(q, x) &= p_\infty, \\
k_l(q, y) &= \mathit{intersect}(q, y), \\
x \oplus y &= \mathit{nearer}(x, y), \\
c_c(q, x, x_\bullet) &= (\mathit{intersect}(q, x - x_\bullet) \equiv p_\infty), \\
k_c(q, x, x_\bullet) &= p_\infty,
\end{aligned}$$

where *intersect* of type $\mathit{Ray} \times \mathit{Object} \to \mathit{Position}$ calculates an intersection position, *nearer* of type $\mathit{Position} \times \mathit{Position} \to \mathit{Position}$ returns the position nearer to the viewpoint, and $p_\infty$ denotes the position of the point at infinity. Each query and *intersect* return $p_\infty$ to denote no intersection.

In photon mapping, photons are stored in the leaves of a $k$-d tree and a query calculates $k$-nearest neighbors of a given point under some bounded range [Jen00]. To calculate a set of $k$-nearest neighbors, a size-bounded mergeable heap is useful. We consider an abstract data type $\mathit{Heap}^k_\alpha$ that denotes a mergeable heap of $k$ elements of type $\alpha$. Its constructor *initHeap* takes an initial set of elements and a weight function $w$ to calculate the weight of an element for sorting. The *merge* operation is defined over heaps that have the same weight function. Since the range of neighbors is bounded, a query point is a ball of type *Ball* that consists of a center position and a radius. Let *Photon* be the type of photons, we can define the types and parameters of iter as follows:

$$\begin{aligned}
\alpha &= \mathit{Rect}_3, \\
\beta &= \mathit{Photon}, \\
\gamma &= \mathit{Ball}, \\
\delta &= \mathit{Heap}^k_{\mathit{Photon}}, \\
c(q, x) &= \mathit{isEmpty}(q \cap x), \\
k_b(q, x) &= \varnothing, \\
k_l(q, p) &= \mathit{initHeap}(\{p\}, \mathit{distFrom}_q), \\
H \oplus H' &= \mathit{merge}(H, H'), \\
c_c(q, x, x_\bullet) &= q \subseteq x_\bullet, \\
k_c(q, x, x_\bullet) &= \varnothing,
\end{aligned}$$

where $\mathit{distFrom}_b$ denotes a function that calculate a distance from the center of ball $b$ to a given photon.

### 12.5.4 Nearest-Neighbor Classifiers

In nearest-neighbor querying by iter, a bounded range of neighbors is important because it is used in top-down pruning. Without this bounded range, a query point would traverse the whole tree and incur a terrible slowdown. While bounded ranges are natural in photon mapping, $k$-nearest-neighbor classifiers, which are extensively used in pattern recognition, do not assume bounded ranges in general.

In the unbounded-range case, we generally perform bottom-up pruning by using an intermediate result of $k$-nearest neighbors. iter cannot deal with this bottom-up pruning. One approach to dealing with this limitation is multi-pass top-down pruning. Specifically, in the first pass, we find a leaf on the basis of a querying point with an empty range and then approximate a range of neighbors from the point(s) in the found leaf. In the second pass, we perform a range-bounded $k$-nearest-neighbor query

with the approximated range. If we do not obtain $k$ neighbors, we retry a range-bounded $k$-nearest-neighbor query by relaxing the current range bound a little. In the after third passes, one-hole range queries suffice for accumulating nearest neighbors.

We can implement bottom-up pruning itself straightforwardly on (buffered) recursively segmented trees. In this case, it will be reasonable to perform bottom-up pruning at the root of each traversed segment on each level. This, however, complicates implementation compared to iter. Because last-level segments would provide reasonably approximated ranges in iterative top-down pruning, it would be sufficient for $k$-nearest-neighbor queries to extend iter in a segment-aware manner.

## 12.6 Related Work

We discuss related work on various aspects.

### 12.6.1 Enhancing Locality in Tree Traversal

Jo et al.'s work [JK11, JK12, JGK13] is the most relevant and similar to our work. Both theirs and ours have the same purpose of providing not application-specific techniques to enhance locality and parallelize, and target almost the same computational patterns. However, while their work aims at automatic optimization, our work aims at abstraction of patterns.

Technically, traversal splicing [JK12] is very similar to the notion of buffered recursively segmented trees. Their traversal splicing divides a traversal into ones on the top half and on the bottom half, records querying points into the horizontal boundary, and then interleaves top-half traversal and bottom-half traversal. Our buffered recursively segmented trees also record querying points of the boundaries of segments. The shapes of boundaries and the ways of recording querying points are, however, different. The boundary in their traversal splicing is not recursive. It means that they assumed balanced trees implicitly. Their traversal splicing records a querying point with a trace from the root and does not record a querying point into multiple nodes on the boundary. That is, their traversal splicing is based on the snapshot/restore of tree traversal. Our approach is based on the divide and conquer of tree traversal. It is advantageous to distributed parallelization of tree traversal.

Last but not least, they did not analyze cache complexity at all. Although we have not analyzed iter itself, we have examined a situation where our techniques would be effective for cache complexity, through a priori analysis of model cases.

### 12.6.2 Cache-Efficient/-Oblivious Trees

Cache-oblivious algorithms [FLPR99] are tuning-free yet cache-efficient ones. Many cache-oblivious data structures, whose operations are cache-oblivious, were developed. Cache-oblivious trees designed for spatial data were, for example, B-trees [MAB00, BDFC05, BFCF$^+$07], $k$-d trees [AADHM03], and mesh layouts [BKTW11]. The main advantage of cache-oblivious ones is portability. Cache-oblivious ones are ready to work efficiently regardless of cache sizes. Moreover, these B-trees and $k$-d trees also guarantee the cache complexity of updating, which our data structures do not guarantee. All in all, these cache-oblivious trees are more useful than ours for a simple single query.

Our data structures are, however, not always poorer than these cache-oblivious ones. For example, a root-to-leaf traversal of B-tree [MAB00, BDFC05, BFCF$^+$07] and $k$-d trees [AADHM03] costs the same $O(\log_L N)$ cache misses as our recursively segmented trees. In addition, it is difficult to compare the cache complexity of iterative tree traversal or iter, for which our buffered recursively segmented trees are designed. In this sense, our data structures are incomparable to the cache-oblivious tree above.

Also developed were cache-efficient (and cache-aware) tree data structures, e.g., sequence heaps [San00] and BVHs [YM06]. The design of buffered recursively segmented trees are similar to that of sequence heaps on the aspects of hierarchical buffering with parametrized fan-outs and buffer sizes.

Cache-efficient/-oblivious trees basically suppose balanced trees or particularly a complete binary trees. A way of balancing trees relies somewhat on the definition of a specific space-partitioning tree. Our approach does not suppose balanced trees. Imbalanced trees are practically simple and/or efficient.

For example, many $N$-body simulations use octree, which can be imbalanced, and imbalanced $k$-d trees were efficient for photon mapping [WGS04]. The main advantage of our approach based on $m$-bridge is genericity regarding the shapes of trees.

### 12.6.3 Iterative Search

In state-space search (or combinatorial search), we can find iterative patterns such as iterative deepening depth-first search, IDA* search, and Monte-Carlo tree search. While the traversal that we deal with is to traverse tree data structures, these traverse state space and the traces of search form trees in hindsight. A state is per se a value used in search and the next states can be calculated from a current state. These are state transition rather than tree traversal. These locality issues are therefore quite different from our iterative tree traversal.

Since state-space search is state transition, the locality in transition table is important. Actually, a distributed parallelization regarding tables [RPBS99] was quite effective for IDA* search [RPBS99] and Monte-Carlo tree search [YKK+11]. As seen from these results, both locality enhancement and parallelization therefore rely strongly on data structures.

### 12.6.4 Data-Parallel Skeletons

In parallel programming, parallel computing patterns are called skeletons [Col89, RG02]. Those which exploit parallelism of data structures are called data-parallel skeletons. Since iter exploits parallelisms of given data structures, we can regard it as a data-parallel skeleton.

What kind of skeletons is iter, then? The map to querying points is a list skeleton [Ski93]. The pruning and reduction of space-partitioning trees can be implemented with tree skeletons [Ski96, GCS94]. Moreover, space-partitioning trees are based on the freedom on dividing a given multidimensional space. This trait in a two-dimensional case is closely relevant to the algebra used in matrix skeletons [EHKT07]. iter is actually a chimera of different data-parallel skeletons. This leads to an interesting observation on skeletons. Even though applications of matrix skeletons and tree ones are very limited, a chimera of them can have important applications.

## 12.7 Conclusion

In this chapter, we have presented an application of $m$-bridging to the design of data structures for iterative tree traversal. Our parallel pattern iter that formulates typical iterative tree traversal can deal on our data structures with important applications.

We plan to implement iter and our data structures and evaluate them experimentally by using important applications.

A further step in future work that we consider is to deal with fast algorithms for the $N$-body problem. Many problems in statistical learning are known to be formalized as a kind of $N$-body problem [GM01]. In Riegel's recent work [Rie13], a wider range of problems were formalized as the generalized $N$-body problem and a fast algorithm for solving it was presented. If we incorporated these with parallel patterns on our data structures, our approach would be more valuable.

# Chapter 13

# Towards Neighborhood Abstractions

In this part, we have investigated parallel programming for neighborhood computations in cache-efficient and divide-and-conquer manners. Unfortunately, we have not built a technical connection between the work in Chapter 11 and that in Chapter 12, which deal with seemingly different computations. Both stencil computation and querying of space-partitioning are, however, conceptually very close.

Stencil computation is usually considered as a regular array-based computation. In this sense, it seems to be out of the scope of tree-based computations. This regularity of stencil computation is, however, derived from a uniform grid. If we adopt an adaptive grid, stencil computation becomes irregular. In fact, adaptive mesh refinement for partial differential equation solvers is considered as adaptive stencils [DS06], and its load balancing uses space-partitioning trees [Mit07]. Moreover, grids can be hierarchical in multigrid methods [BHM00]. In this case, stencil computation becomes a tree-based computation.

In querying of space-partitioning trees, we find neighbors by visiting spatial nodes. We can also consider this visited nodes as neighbors, which have various granularity. In this sense, a query is a tree-shaped stencil. Besides, in the domain of $N$-body problems, fast multipole methods (FMMs), which are faster algorithms based on space-partitioning trees, were developed. The original FMM [GR87] performs a typical stencil computation at each spatial level.

These connections between stencil computation and space-partitioning trees are derived from physical laws. Physical contributions such as gravity are inversely proportional to distance in general. It is natural to use neighbors for calculating their approximations. Since we always calculate approximations of model formulae in scientific computing on the basis of numerical analysis, use of neighborhood is primordial. Then, to improve the precision of approximations in general, we have to calculate contributions of a far distance. Hierarchical subspaces, which space-partitioning trees represent, enable us to interpret faraway areas adaptively as neighbors. Use of space-partitioning trees is therefore reasonable to improve the precision of approximations.

Because of this natural connection, it is desired to unify abstractions of stencil computation and querying of space-partitioning trees. We have been pursued a unified abstraction for various neighborhood computations. At the beginning, we conjectured that trees were useful for such a unified abstraction. Through the work in this part, we, however, found trees to be inadequate to abstract neighborhood computations because a tree structure is merely an axis of a neighborhood subspace. We have to focus on the formalization of neighborhood rather than overall iterative and/or recursive computations. In fact, from the perspective of abstractions of neighborhood, Chapter 11 formalizes neighborhood as constant band matrices and Chapter 12 formalizes it as a root-to-leaf path in space-partitioning trees. Since both are far different, it is reasonably difficult to build a technical connection between both.

If we focus on the formalization of neighborhood, a unified abstraction for parallel neighborhood computations is still difficult to design. This is because the structures of neighborhood affect global recursions of individual neighborhood computations. For example, Callahan-Kosaraju algorithms [CK95, Cal93] construct neighborhood as well-separated pairs on space-partitioning trees. We can represent these pairs as side links among nodes. However, the construction of these pairs is not a recursion on trees but a recursion on pairs of subspaces. In fact, it contains a peculiar shuffling of siblings from the viewpoint of tree computations. Because the global recursion of a whole computation is of key importance of load

balancing, the variety of global recursions depending on kinds of neighborhood makes it difficult to design abstractions. We therefore leave it for future work.

# Chapter 14

# Conclusion

## 14.1 Summary of Contributions

In this dissertation, we have dealt with parallel programming with trees in a divide-and-conquer manner. Our main contributions in this work are summarized as follows:

- We have designed an iterator-based interface between tree skeletons and tree-structure implementation, and have developed a tree skeleton library loosely coupled between both by using our interface. We have demonstrated the benefits of its flexibility experimentally. (Chapter 3)
- We have developed a parallelizer that transforms sequential recursive functions in C into tree skeleton calls with operators based on the formalization by Matsuzaki at al. [MHT06]. It hides the complicated API of tree skeletons from programmers and brings the benefits of tree skeletons with no burden on programmers. (Chapter 4)
- We have developed a novel syntax-directed divide-and-conquer method of data-flow analysis based on Tarjan's formalization [Tar81a] and Rosen's high-level approach [Ros77, Ros80]. It can deal with arbitrary goto/label statements. We have demonstrated the feasibility and scalability of our method experimentally through prototype implementations on a C compiler. (Chapter 7)
- We have developed a syntax-directed divide-and-conquer method for constructing value graphs on the basis of a functional formalization of value graphs and Rosen's high-level approach [Ros77, Ros80]. It tames goto/label statements troublesome in $\phi$-function placement. (Chapter 8)
- We have developed a linear algebraic approach to optimizing stencil computation. We have presented its concept, techniques to implement it, and its effects on asymptotic complexities. We demonstrated its performance gain experimentally through a prototype library. (Chapter 11)
- We have developed techniques for enhancing the asymptotic cache complexity of iterative traversal of space-partitioning trees, on the basis of its skeletal formalization and $m$-bridging [Rei93]. We have explored the applicability of our approach in practically important problems. (Chapter 12)

## 14.2 Retrospection

As concluding remarks of this dissertation, we describe the retrospection on our work and summarize our observations from practice.

First of all, we should clarify the meaning of *structured* approaches that we have pursued. As seen from that the notion of structured programming [BJ66, Dij68, Knu74, DDH72] was controversial, to define the notion of "structured" formally and reasonably is difficult. We think that a little ambiguity of "structured" is unavoidable. Yet, we intentionally use "structured" because we consider it to be the principle of programming and abstraction. "A structured manner" in this dissertation roughly means

to use or obey some pattern. This pattern would be highly abstract or a little ambiguous. We think that attitudes towards patterns rather than patterns themselves are of huge significance. For example, it is important to discover patterns, to apply patterns, to compose patterns, and especially, to restrict oneself to some pattern. In other words, a disciplined attitude through patterns is of true significance. To pursue a structured approach is therefore to investigate how to restrict oneself to a pattern or style reasonable in concerns and/or domains.

Our primary concern is load balancing for parallel computing. Computations easy to load balance are of crucial importance for parallel programs portable among different parallel machines. We have adopted the divide-and-conquer paradigm for the primary pattern in this concern because divide-and-conquer algorithms are ready both for parallelization and load balancing. We have restricted ourselves to the divide-and-conquer paradigm. How "structured" our approaches are is therefore how much part is in a divide-and-conquer manner.

We have been focusing on use of trees because of its versatility in programming and its affinity with the divide-and-conquer paradigm. Our work started from tree skeletons (Chapter 2) because tree skeletons were indeed patterns that enabled divide-and-conquer load balancing and because the work on list skeletons achieved solid results both in theory and practice.

Our work, in fact, resulted in several improvements on usability of tree skeletons (Chapters 3 and 4). Yet, tree skeletons did not become useful for actual problems. By rethinking list skeletons, we have noticed that tree skeletons are unnatural and inherently not intuitive for programming with trees (Chapter 5). A tree is usually a representation equipped with an interpretation. The interpretation of a tree often determines an essential structure of the tree and operators used with the tree. Because tree skeletons operate trees without any interpretation, it is natural that they are difficult to use. We therefore separate away from tree skeletons.

Instead of tree skeletons, we next adopted syntax-directed programming. This is to describe an interpretation of trees in their syntax and to define calculations on their syntax, and to perform load balancing by utilizing the interpretation of trees. This is a typical style in functional programming, has been used to formalize list skeletons, and obeys by definition the divide-and-conquer paradigm. We selected AST-based program analysis as a case study of syntax-directed programming.

Our work on AST-based program analysis resulted successfully in syntax-directed divide-and-conquer methods (Chapters 7 and 8). We also developed how to deal with goto-derived data flow, which is a typical irregularity in syntax-directed computations. We have two AST-based methods on the same language. This result justified our separation from tree skeletons. The interpretation of trees is more important than tree structure.

Because neighborhood computations are very popular in various domains, we have dealt with cache-efficient divide-and-conquer programming for neighborhood computations (Chapters 11 and 12). Unfortunately, we have not developed a unified tree-based abstraction for various neighborhood computations because the abstraction of neighborhood subspace are inherently beyond trees (Chapter 13). However, by limiting the interpretation of trees to space partitioning and formalizing neighborhood as a root-to-leaf path in trees, we have found another usage of segmented trees (Chapter 12). In this case, tree structures have been, in fact, non-essential either to specification or abstraction. It suggest that what we should consider truly is not the structure of trees but the interpretation of trees.

In summary, we have learnt from practice the following lessons:

- The interpretation of trees is an essential issue in programming with trees.
- Syntax-directed programming is both reasonable and useful for parallel programming.
- Tree structures can be non-essential either to specification or abstraction even in application domains in which trees are extensively used.

Although these are not outstandingly novel, these are reasonable; these are realities.

# Bibliography


[AADHM03] Pankaj K. Agarwal, Lars Arge, Andrew Danner, and Bryan Holland-Minkley. Cache-Oblivious Data Structures for Orthogonal Range Searching. In *Proc. SCG '03*, pages 237–245. ACM, 2003.

[AC76] Frances E. Allen and John Cocke. A Program Data Flow Analysis Procedure. *Commun. ACM*, 19(3):137–147, 1976.

[ADKP89] Karl R. Abrahamson, Norm Dadoun, David G. Kirkpatrick, and Teresa M. Przytycka. A Simple Parallel Tree Contraction Algorithm. *J. Algorithms*, 10(2):287–302, 1989.

[AH00] John Aycock and R. Nigel Horspool. Simple Generation of Static Single-Assignment Form. In *Compiler Construction (Proc. CC '00)*, volume 3923 of *Lecture Notes in Computer Science*, pages 110–124. Springer, 2000.

[AK01] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers, 2001.

[AKNR12] Aws Albarghouthi, Rahul Kumar, Aditya V. Nori, and Sriram K. Rajamani. Parallelizing Top-Down Interprocedural Analyses. In *Proc. PLDI '12*, pages 217–228. ACM, 2012.

[AS07] Shai Avidan and Ariel Shamir. Seam Carving for Content-Aware Image Resizing. In *Proc. SIGGRAPH '07*. ACM, 2007.

[AWZ88] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting Equality of Variables in Programs. In *Proc. POPL '88*, pages 1–11. ACM, 1988.

[Bak77] Brenda S. Baker. An Algorithm for Structuring Flowgraphs. *J. ACM*, 24(1):98–120, 1977.

[BBH+13] Matthias Braun, Sebastian Buchwald, Sebastian Hack, Roland Leißa, Christoph Mallon, and Andreas Zwinkau. Simple and Efficient Construction of Static Single Assignment Form. In *Compiler Construction (Proc. CC '13)*, volume 7791 of *Lecture Notes in Computer Science*, pages 102–122. Springer, 2013.

[BCH+94] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a Portable Nested Data-Parallel Language. *J. Parallel Distrib. Comput.*, 21(1):4–14, 1994.

[BCHS98] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical Improvements to the Construction and Destruction of Static Single Assignment Form. *Softw. Pract. Exp.*, 28(8):859–881, 1998.

[BCS97] Preston Briggs, Keith D. Cooper, and L. Taylor Simpson. Value Numbering. *Softw. Pract. Exp.*, 27(6):701–724, 1997.

[BDFC05] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-Oblivious B-Trees. *SIAM J. Comput.*, 35(2):341–358, 2005.

[BFCF+07] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. Cache-Oblivious Streaming B-trees. In *Proc. SPAA '07*, pages 81–92. ACM, 2007.

[BFGS12] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. Internally Deterministic Parallel Algorithms Can Be Fast. In *Proc. PPoPP '12*, pages 181–192. ACM, 2012.

[BFR+13] Lars Bergstrom, Matthew Fluet, Mike Rainey, John Reppy, Stephen Rosen, and Adam Shaw. Data-Only Flattening for Nested Data Parallelism. In *Proc. PPoPP '13*, pages 81–92. ACM, 2013.

[BGS10] Guy E. Blelloch, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Low Depth Cache-Oblivious Algorithms. In *Proc. SPAA '10*, pages 189–199. ACM, 2010.

[BHM00] William L. Briggs, Van Emden Henson, and Steve F. McCormick. *A Multigrid Tutorial*. SIAM, 2nd edition, 2000.

[BHMS91] Mark Bromley, Steven Heller, Tim McNerney, and Guy L. Steele Jr. Fortran at ten gigaflops: The connection machine convolution compiler. In *Proc. PLDI '91*, pages 145–156. ACM, 1991.

[BHRS08] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A Practical Automatic Polyhedral Program Optimization System. In *Proc. PLDI '08*, pages 101–113. ACM, 2008.

[BHT+10] Muthu Manikandan Baskaran, Albert Hartono, Sanket Tavarageri, Tom Henretty, J. Ramanujam, and P. Sadayappan. Parameterized Tiling Revisited. In *Proc. CGO '10*, pages 200–209. ACM, 2010.

[Bir87] Richard 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 3–42. Springer-Verlag, 1987.

[BJ66] Corrado Böhm and Giuseppe Jacopini. Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules. *Commun. ACM*, 9(5):366–371, 1966.

[BKTW11] Michael A. Bender, Bradley C. Kuszmaul, Shang-Hua Teng, and Kebin Wang. Optimal Cache-Oblivious Mesh Layouts. *Theory Comput. Syst.*, 48(2):269–296, 2011.

[Ble90] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. The MIT Press, 1990.

[Ble93] Guy E. Blelloch. Prefix Sums and Their Applications. In *Synthesis of Parallel Algorithms*, chapter 1. Morgan Kaufmann Publishers, 1993.

[BM94] Marc M. Brandis and Hanspeter Mössenböck. Single-Pass Generation of Static Single-Assignment Form for Structured Languages. *ACM Trans. Program. Lang. Syst.*, 16(6):1684–1698, 1994.

[BMO90] Robert A. Ballance, Arthur B. Maccabe, and Karl J. Ottenstein. The Program Dependence Web: A Representation Supporting Control-, Data-, and Demand-Driven Interpretation of Imperative Languages. In *Proc. PLDI '90*, pages 257–271. ACM, 1990.

[BP03] Gianfranco Bilardi and Keshav Pingali. Algorithms for Computing the Static Single Assignment Form. *J. ACM*, 50(3):375–425, 2003.

[Cal93] Paul B. Callahan. Optimal Parallel All-Nearest-Neighbors Using the Well-Separated Pair Decomposition. In *Proc. FOCS '93*, pages 332–340. IEEE, 1993.

[CCF91] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic Construction of Sparse Data Flow Evaluation Graphs. In *Proc. POPL '91*, pages 55–56. ACM, 1991.

[Cel12] Joe Celko. *Joe Celko's Trees and Hierarchies in SQL for Smarties.* Morgan Kaufmann Publishers, 2 edition, 2012.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. EFficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[CK95] Paul B. Callahan and S. Rao Kosaraju. A Decomposition of Multidimensional Point Sets with Applications to $k$-Nearest-Neighbors and $n$-Body Potential Fields. *J. ACM*, 42(1):67–90, 1995.

[Col89] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* Research Monographs in Parallel and Distributed Computing. MIT Press, 1989.

[CRP+10] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. FlumeJava: Easy, Efficient Data-Parallel Pipelines. In *Proc. PLDI '10*, pages 363–375. ACM, 2010.

[CS70] John Cocke and Jacob T. Schwartz. Programming Languages and Their Compilers: Preliminary Notes. Technical report, Courant Institute of Mathematical Sciences, New York Univ., 1970. Second Version.

[DDH72] Ole-Johan Dahl, Edsger W. Dijkstra, and C. A. R. Hoare, editors. *Structured Programming.* Academic Press, 1972.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. OSDI '04*, pages 137–150. USENIX, 2004.

[DGTY95] John Darlington, Yi-ke Guo, Hing Wing To, and Jin Yang. Parallel Skeletons for Structured Composition. In *Proc. PPoPP '95*, pages 19–28. ACM, 1995.

[Dij68] Edsger W. Dijkstra. Letters to the Editor: Go to Statement Considered Harmful. *Commun. ACM*, 11(3):147–148, 1968.

[DMV+08] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David A. Patterson, John Shalf, and Katherine A. Yelick. Stencil Computation Optimization and Auto-tuning on State-of-the-Art Multicore Architectures. In *Proc. SC '08*, pages 4:1–4:12. IEEE, 2008.

[DS06] H. Ding and C. Shu. A stencil adaptive algorithm for finite difference solution of incompressible viscous flows. *J. Comput. Phys.*, 214:397–420, 2006.

[DST80] Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the Common Subexpression Problem. *J. ACM*, 27(4):758–771, 1980.

[EFH12] Kento Emoto, Sebastian Fischer, and Zhenjiang Hu. Filter-embedding semiring fusion for programming with MapReduce. *Formal Asp. Comput.*, 24(4-6):623–645, 2012.

[EH07] Torbjörn Ekman and Görel Hedin. The JastAdd system: modular extensible compiler construction. *Sci. Comput. Program.*, 69(1–3):14–26, 2007.

[EHKT07] Kento Emoto, Zhenjiang Hu, Kazuhiko Kakehi, and Masato Takeichi. A Compositional Framework for Developing Parallel Programs on Two-Dimensional Arrays. *Int. J. Parallel Program.*, 35(6):615–658, 2007.

[EI12] Kento Emoto and Hiroto Imachi. Parallel Tree Reduction on MapReduce. In *Proc. ICCS '12*, volume 9 of *Procedia Computer Science*, pages 1827–1836. Elsevier, 2012.

[EM14] Kento Emoto and Kiminori Matsuzaki. An Automatic Fusion Mechanism for Variable-Length List Skeletons in SkeTo. *Int. J. Parallel Program.*, 42(4):546–563, 2014. In Proc. HLPP '13.

[FG94] Allan L. Fisher and Anwar M. Ghuloum. Parallelizing Complex Scans and Reductions. In *Proc. PLDI '94*, pages 135–146. ACM, 1994.

[FLPR99] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-Oblivious Algorithms. In *Proc. FOCS '99*, pages 285–297. IEEE, 1999.

[FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Proc. PLDI '98*, pages 212–223. ACM, 1998.

[FS05] Matteo Frigo and Volker Strumpen. Cache oblivious stencil computations. In *Proc. ICS '05*, pages 361–366. ACM, 2005.

[FS06] Matteo Frigo and Volker Strumpen. The Cache Complexity of Multithreaded Cache Oblivious Algorithms. In *Proc. SPAA '06*, pages 271–280. ACM, 2006.

[FS07] Matteo Frigo and Volker Strumpen. The Memory Behavior of Cache Oblivious Stencil Computations. *J. Supercomput.*, 39(2):93–112, 2007.

[GCS94] Jeremy Gibbons, Wentong Cai, and David B. Skillicorn. Efficient Parallel Algorithms for Tree Accumulations. *Sci. Comput. Program.*, 23(1):1–18, 1994.

[GF64] Bernard A. Galler and Michael J. Fisher. An Improved Equivalence Algorithm. *Commun. ACM*, 7(5):301–303, 1964.

[GGL12] Tobias Grosser, Armin Größlinger, and Christian Lengauer. Polly—performing polyhedral optimizations on a low-level intermediate representation. *Parallel Process. Lett.*, 22(4), 2012.

[GL05] Douglas Gregor and Andrew Lumsdaine. Lifting Sequential Graph Algorithms for DistributedMemory Parallel Computation. In *Proc. OOPSLA '05*, pages 423–437. ACM, 2005.

[GM01] Alexander G. Gray and Andrew W. Moore. 'N-Body' Problems in Statistical Learning. In *Advances in Neural Information Processing Systems 13 (NIPS 2000)*, pages 521–527, 2001.

[GR87] Leslie Greengard and Vladimir Rokhlin. A Fast Algorithm for Particle Simulations. *J. Comput. Phys.*, 73(2):325–348, 1987.

[GVL10] Horacio González-Vélez and Mario Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Softw. Pract. Exp.*, 40(12):1135–1160, 2010.

[HFAR13] Harshvardhan, Adam Fidel, Nancy M. Amato, and Lawrence Rauchwerger. The STAPL Parallel Graph Library. In *Languages and Compilers for Parallel Computing (LCPC '12, Revised Selected Papers)*, volume 7760 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 2013.

[HPP09] Mary W. Hall, David A. Padua, and Keshav Pingali. Compiler Research: The Next 50 Years. *Commun. ACM*, 52(2):60–67, 2009.

[HSP+11] Tom Henretty, Kevin Stock, Louis-Noël Pouchet, Franz Franchetti, J. Ramanujam, and P. Sadayappan. Data Layout Transformation for Stencil Computations on Short-Vector SIMD Architectures. In *Compiler Construction (Proc. CC '11)*, volume 6601 of *Lecture Notes in Computer Science*, pages 225–245. Springer, 2011.

[HTI99] Zhenjiang Hu, Masato Takeichi, and Hideya Iwasaki. Diffusion: Calculating Efficient Parallel Programs. In *Proc. PEPM '99*, pages 85–94. ACM, 1999.

[IKF05] Yusuke Ichikawa, Zenjiro Konishi, and Yoshihiko Futamura. Recursion Removal from Recursive Programs with Only One Descent Function. *IEICE Trans. Info. Sys.*, E88-D(2):187–196, 2005.

[Jen00] Henrik Wann Jensen. A Practical Guide to Global Illumination Using Photon Mapping. SIGGRAPH 2000 Course 8, July 2000. `http://graphics.stanford.edu/courses/cs348b-01/course8.pdf`.

[JGK13] Youngjoon Jo, Michael Goldfarb, and Milind Kulkarni. Automatic Vectorization of Tree Traversals. In *Proc. PACT '13*, pages 363–374. IEEE, 2013.

[JK11] Youngjoon Jo and Milind Kulkarni. Enhancing Locality for Recursive Traversals of Recursive Structures. In *Proc. OOPSLA '11*, pages 463–482. ACM, 2011.

[JK12] Youngjoon Jo and Milind Kulkarni. Automatically Enhancing Locality for Tree Traversals with Traversal Splicing. In *Proc. OOPSLA '12*, pages 355–374. ACM, 2012.

[KBB+07] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Effective Automatic Parallelization of Stencil Computations. In *Proc. PLDI '07*, pages 235–244, 2007.

[KBI+09] Milind Kulkarni, Martin Burtscher, Rajeshkar Inkulu, Keshav Pingali, and Calin Casçaval. How Much Parallelism is There in Irregular Applications? In *Proc. PPoPP '09*, pages 3–14. ACM, 2009.

[KCL+12] Gabriele Keller, Manuel M. T. Chakravarty, Roman Leshchinskiy, Ben Lippmeier, and Simon Peyton Jones. Vectorisation avoidance. In *Proc. Haskell '12*, pages 37–48. ACM, 2012.

[KD94] Uday P. Khedker and Dhananjay M. Dhamdhere. A Generalized Theory of Bit Vector Data Flow Analysis. *ACM Trans. Program. Lang. Syst.*, 16(5):1472–1511, 1994.

[KGS94] Robert Kramer, Rajiv Gupta, and Mary Lou Soffa. The Combining DAG: A Technique for Parallel Data Flow Analysis. *IEEE Trans. Parallel Distr. Syst.*, 5(8):805–813, 1994.

[Kil73] Gary A. Kildall. A Unified Approach to Global Program Optimization. In *Proc. POPL '73*, pages 194–206. ACM, 1973.

[KME07] Kazuhiko Kakehi, Kiminori Matsuzaki, and Kento Emoto. Efficient Parallel Tree Reductions on Distributed Memory Environments. In *Computational Science – ICCS 2007*, volume 4488 of *Lecture Notes in Computer Science*, pages 601–608. Springer, 2007.

[Knu68] Donald E. Knuth. Semantics of Context-Free Languages. *Math. Syst. Theory*, 2(2):127–145, 1968.

[Knu74] Donald E. Knuth. Structured Programming with go to Statements. *ACM Comput. Surv.*, 6(4):261–301, 1974.

[KS98] Kathleen Knobe and Vivek Sarkar. Array SSA Form and Its Use in Parallelization. In *Proc. POPL '98*, pages 107–120. ACM, 1998.

[LAAK06] Beatrice Luca, Stefan Andrei, Hugh Anderson, and Siau-Cheng Khoo. Program Transformation by Solving Recurrences. In *Proc. PEPM '06*, pages 121–129. ACM, 2006.

[LEH14] Yu Liu, Kento Emoto, and Zhenjiang Hu. A Generate-Test-Aggregate Parallel Programming Library. *Parallel Comput.*, 40(2):116–135, 2014.

[LKK13] Jonathan Lifflander, Sriram Krishnamoorthy, and Laxmikant V. Kale. Steal Tree: Low-Overhead Tracing of Work Stealing Schedulers. In *Proc. PLDI '13*, pages 507–518. ACM, 2013.

[LRF95] Yong-Fong Lee, Barbara G. Ryder, and Marc E. Fiuczynski. Region Analysis: A Parallel Elimination Method for Data Flow Analysis. *IEEE Software Eng.*, 21(11):913–926, 1995.

[LTA03] Andrew A. Lamb, William Thies, and Saman Amarasinghe. Linear Analysis and Optimization of Stream Programs. In *Proc. PLDI '03*, pages 12–25. ACM, 2003.

[MAB00] Martin Farach-Colton Michael A. Bender, Erik D. Demaine. Cache-Oblivious B-Trees. In *Proc. FOCS '00*, pages 399–409. IEEE, 2000.

[Mat07a] Kiminori Matsuzaki. Implementation of Tree Accumulations on Distributed-Memory Parallel Computers. In *Computational Science – ICCS 2007*, volume 4488 of *Lecture Notes in Computer Science*, pages 609–616. Springer, 2007.

[Mat07b] Kiminori Matsuzaki. *Parallel Programming with Tree Skeletons*. PhD thesis, The University of Tokyo, 2007.

[ME10] Kiminori Matsuzaki and Kento Emoto. Implementing Fusion-Equipped Parallel Skeletons by Expression Templates. In *Implementation and Application of Functional Languages (IFL '09, Revised Selected Papers)*, volume 6041 of *Lecture Notes in Computer Science*, pages 72–89. Springer, 2010.

[MFS79] Ronald J. Mintz, Gerald A. Fisher, and Micha Sharir. The design of a global optimizer. In *Proc. SIGPLAN '79 Symp. Compiler Construction*, pages 226–234. ACM, 1979.

[MGL+10] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive Analysis of Web-Scale Datasets. *VLDB Journal*, 3(1):330–339, 2010.

[MHT06] Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Towards Automatic Parallelization of Tree Reductions in Dynamic Programming. In *Proc. SPAA '06*, pages 39–48. ACM, 2006.

[MHT08] Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Derivation of Parallel Programs for Maximum Marking Problems on Lists. *IPSJ Trans. PRO*, 49(SIG 3 (PRO 36)):16–27, 2008. In Japanese.

[Mit07] William F. Mitchell. A refinement-tree based partitioning method for dynamic load balancing with adaptively refined grids. *J. Parallel Distr. Comput.*, 67(4):417–429, 2007.

[MLBP12] Mario Méndez-Lojo, Martin Burtscher, and Keshav Pingali. A GPU Implementation of Inclusion-based Points-to Analysis. In *Proc. PPoPP '12*, pages 107–116. ACM, 2012.

[MLMP10] Mario Méndez-Lojo, Augustine Mathew, and Keshav Pingali. Parallel Inclusion-based Points-to Analysis. In *Proc. OOPSLA '10*, pages 428–443. ACM, 2010.

[MM10] Akimasa Morihata and Kiminori Matsuzaki. Automatic Parallelization of Recursive Functions using Quantifier Elimination. In *Functional and Logic Programming (Proc. FLOPS '10)*, volume 6009 of *Lecture Notes in Computer Science*, pages 321–336, 2010.

[MM11a] Akimasa Morihata and Kiminori Matsuzaki. A Practical Tree Contraction Algorithm for Parallel Skeletons on Trees of Unbounded Degree. In *Proc. ICCS '11*, volume 4 of *Procedia Computer Science*, pages 7–16. Elsevier, 2011.

[MM11b] Akimasa Morihata and Kiminori Matsuzaki. Balanced Trees Inhabiting Functional Parallel Programming. In *Proc. ICFP '11*, pages 117–128. ACM, 2011.

[MM14] Kiminori Matsuzaki and Reina Miyazaki. Parallel Tree Accumulations on MapReduce. In *Proc. HLPP '14*, pages 31–50, 2014.

[MMHT09] Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. The Third Homomorphism Theorem on Trees. In *Proc. POPL '09*, pages 177–185. ACM, 2009.

[MMM+07] Kazutaka Morita, Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Automatic Inversion Generates Divide-and-Conquer Parallel Programs. In *Proc. PLDI '07*, pages 146–155. ACM, 2007.

[Mor11] Akimasa Morihata. Macro Tree Transformations of Linear Size Increase Achieve Cost-Optimal Parallelism. In *Programming Languages and Systems (Proc. APLAS '11)*, volume 7078 of *Lecture Notes in Computer Science*, pages 204–219. Springer, 2011.

[Mor12] Akimasa Morihata. Parallel Evaluation of Macro Tree Transducers with Parallel Tree Contraction. In *Proc. 29th Conference of JSSST*, 2012. In Japanese.

[Mor13] Akimasa Morihata. A Short Cut to Parallelization Theorems. In *Proc. ICFP '13*, pages 245–256. ACM, 2013.

[MR85] Gary L. Miller and John H. Reif. Parallel Tree Contraction and Its Application. In *Proc. FOCS '85*, pages 478–489. IEEE, 1985.

[MR90] Thomas J. Marlowe and Barbara G. Ryder. Properties of data flow frameworks. *Acta Inform.*, 28(2):121–163, 1990.

[MW92] Frank Mueller and David B. Whalley. Avoiding Unconditional Jumps by Code Replication. In *Proc. PLDI '92*, pages 322–330. ACM, 1992.

[MW95] Frank Mueller and David B. Whalley. Avoiding Conditional Branches by Code Replication. In *Proc. PLDI '95*, pages 56–66. ACM, 1995.

[MW99] John McCalpin and David Wonnacott. Time Skewing: A Value-Based Approach to Optimizing for Memory Locality. Technical Report DCS-TR-379, Dept. of Computer Science, Rutgers University, 1999.

[NEM+07] Yoshiki Nomura, Kento Emoto, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Parallelization of XPath Queries with Tree Skeletons. *Computer Software*, 24(3):51–62, 2007. In Japanese.

[NG13] Vaivaswatha Nagaraj and R. Govindarajan. Parallel Flow-Sensitive Pointer Analysis by Graph-Rewriting. In *Proc. PACT '13*, pages 19–28. IEEE, 2013.

[OG09] Daniel A. Orozco and Guang R. Gao. Mapping the FDTD Application to Many-Core Chip Architectures. In *Proc. ICPP '09*, pages 309–316. IEEE, 2009.

[PBRO11] Aleksandar Prokopec, Phil Bagwell, Tiark Rompf, and Martin Odersky. A Generic Parallel Collection Framework. In *Euro-Par 2011 Parallel Processing*, volume 6853 of *Lecture Notes in Computer Science*, pages 136–147. Springer, 2011.

[PDGQ05] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the Data: Parallel Analysis with Sawzall. *Sci. Program.*, 13(4):277–298, 2005.

[PRMH11] Tarun Prabhu, Shreyas Ramalingam, Matthew Might, and Mary Hall. EigenCFA: Accelerating Flow Analysis with GPUs. In *Proc. POPL '11*, pages 511–522. ACM, 2011.

[Rei78] John H. Reif. Symbolic Program Analysis in Almost Linear Time. In *Proc. POPL '78*, pages 76–83. ACM, 1978.

[Rei93] John H. Reif. List Ranking and Parallel Tree Contraction. In *Synthesis of Parallel Algorithms*, chapter 3. Morgan Kaufmann Publishers, 1993.

[RF00] Xavier Redon and Paul Feautrier. Detection of Scans in the Polytope Model. *Parallel Algorithms Appl.*, 15(3–4):229–263, 2000.

[RG02] Fethi A. Rabhi and Sergei Gorlatch, editors. *Patterns and Skeleotns for Parallel and Distributed Computing*. Springer, 2002.

[Rie13] Ryan Nelson Riegel. *Generalized N-Body Problems: A Framework for Scalable Computation*. PhD thesis, School of Computational Science and Engineering, Georgia Institute of Technology, December 2013.

[RJDH14] Christopher Rodrigues, Thomas Jablin, Abdul Dakkak, and Wen-Mei Hwu. Triolet: A Programming System That Unifies Algorithmic Skeleton Interfaces for High-performance Cluster Computing. In *Proc. PPoPP '14*, pages 247–258. ACM, 2014.

[RL77] John H. Reif and Harry R. Lewis. Symbolic Evaluation and the Global Value Graph. In *Proc. POPL '77*, pages 104–118. ACM, 1977.

[RL86] John H. Reif and Harry R. Lewis. Efficient Symbolic Analysis of Programs. *J. Comput. Syst. Sci.*, 32(3):280–314, 1986.

[RL11] Jonathan Rodriguez and Ondřej Lhoták. Actor-Based Parallel Dataflow Analysis. In *Compiler Construction (Proc. CC '11)*, volume 6601 of *Lecture Notes in Computer Science*, pages 179–197. Springer, 2011.

[RMCKB97] Gerald Roth, John Mellor-Crummey, Ken Kennedy, and R. Gregg Brickner. Compiling stencils in high performance fortran. In *Proc. SC '97*, pages 1–20. ACM, 1997.

[Ros77] Barry K. Rosen. High-Level Data Flow Analysis. *Commun. ACM*, 20(10):712–724, 1977.

[Ros80] Barry K. Rosen. Monoids for Rapid Data Flow Analysis. *SIAM J. Comput.*, 9(1):159–196, 1980.

[RP86] Barbara G. Ryder and Marvin C. Paull. Elimination Algorithms for Data Flow Analysis. *ACM Comput. Surv.*, 18(3):277–316, 1986.

[RPBS99] John W. Romein, Aske Plaat, Henri E. Bal, and Jonathan Schaeffer. Transposition Table Driven Work Scheduling in Distributed Search. In *Proc. IAAI '99*, pages 725–731. AAAI, 1999.

[RT82] John H. Reif and Robert Endre Tarjan. Symbolic Program Analysis in Almost-Linear Time. *SIAM J. Comput.*, 11(1):81–93, 1982.

[RWZ88] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global Value Numbers and Redundant Computations. In *Proc. POPL '88*, pages 12–27. ACM, 1988.

[San00] Peter Sanders. Fast Priority Queues for Cached Memory. *ACM J. Exp. Algalgorithm.*, 5:7:1–7:25, 2000.

[Sat14a] Shigeyuki Sato. Locality-Aware Parallel Iterative Tree Traversal. In *Proc. 31st Conference of JSSST*, 2014. Non-refereed, unpublished.

[Sat14b] Shigeyuki Sato. Syntax-Directed Contraction of Value Graphs. In *Proc. 16th JSSST Workshop on Programming and Programming Languages (PPL2014)*, 2014. In Japanese, unpublished.

[Sha80] Micha Sharir. Structural Analysis: A New Approach to Flow Analysis in Optimizing Compilers. *Computer Languages*, 5(3–4):141–153, 1980.

[SI11a] Shigeyuki Sato and Hideya Iwasaki. Automatic Parallelization via Matrix Multiplication. In *Proc. PLDI '11*, pages 470–479. ACM, 2011.

[SI11b] Shigeyuki Sato and Hideya Iwasaki. Time Contraction: Simplifying Locality Optimization for Stencil Computation. In *Proc. 28th Conference of JSSST*, 2011. Non-refereed, unpublished.

[Ski93] David B. Skillicorn. The Bird-Meertens Formalism as a Parallel Model. In *Software for Parallel Computation*, volume 106 of *NATO ASI Series*, pages 120–133. Springer, 1993.

[Ski96] David B. Skillicorn. Parallel Implementation of Tree Skeletons. *J. Parallel Distrib. Comput.*, 39(2):115–125, 1996.

[Ski97] David B. Skillicorn. Structured Parallel Computation in Structured Documents. *J. Univers. Comput. Sci.*, 3(1):42–68, 1997.

[SM13] Shigeyuki Sato and Kiminori Matsuzaki. An Operator Generator for Skeletal Programming on Trees. *IPSJ Trans. PRO*, 6(4):38–49, 2013. In Japanese.

[SM14a] Shigeyuki Sato and Kiminori Matsuzaki. A Generic Implementation of Tree Skeletons. In *Proc. HLPP '14*, pages 51–72, 2014.

[SM14b] Shigeyuki Sato and Akimasa Morihata. Syntax-Directed Divide-and-Conquer Data-Flow Analysis. In *Programming Languages and Systems (Proc. APLAS '14)*, volume 8858 of *Lecture Notes in Computer Science*, pages 392–407. Springer, 2014.

[SNM07] Don Syme, Gregory Neverov, and James Margetson. Extensible Pattern Matching Via a Lightweight Language Extension. In *Proc. ICFP '07*, pages 29–40. ACM, 2007.

[SP81] Micha Sharir and Amir Pnueli. *Two Approaches to Inter-Procedural Data-Flow Analysis.* Prentice-Hall, 1981.

[SS69] Robert M. Shapiro and Harry Saint. The Representation of Algorithms. Technical report, Applied Data Research, Inc., 1969.

[SS12] Bjarne Stroustrup and Andrew Sutton. A Concept Design for the STL. Technical Report N3351=12-0041, JTC1/SC22/WG21 – The C++ Standards Committee, 2012.

[SW12] James Stanier and Des Watson. A study of irreducibility in C programs. *Softw. Pract. Exp.*, 42(1):117–130, 2012.

[Tar81a] Robert Endre Tarjan. A Unified Approach to Path Problems. *J. ACM*, 28(3):577–593, 1981.

[Tar81b] Robert Endre Tarjan. Fast Algorithms for Solving Path Problems. *J. ACM*, 28(3):594–614, 1981.

[TBF+11] Gabriel Tanase, Antal Buss, Adam Fidel, Harshvardhan, Ioannis Papadopoulos, Olga Pearce, Timmie G. Smith, Nathan Thomas, Xiabing Xu, Nedal Mourad, Jeremy Vu, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger. The STAPL Parallel Container Framework. In *Proc. PPoPP '11*, pages 235–246. ACM, 2011.

[TCK+11] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. The Pochoir Stencil Compiler. In *Proc. SPAA '11*, pages 117–128. ACM, 2011.

[TSTL09] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality Saturation: A New Approach to Optimization. In *Proc. POPL '09*, pages 264–276, 2009.

[TSTL10] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Translating between PEGs and CFGs. Technical report, University of California, San Diego, October 2010.

[TSTL11] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality Saturation: A New Approach to Optimization. *Log. Meth. Comput. Sci.*, 7(1), 2011.

[Wai90] William M. Waite. Use of Attribute Grammars in Compiler Construction. In *Proc. WAGA*, pages 255–265, 1990.

[WBGK10] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: An extensible attribute grammar system. *Sci. Comput. Program.*, 75(1–2):39–54, 2010.

[WdMBK02] Eric Van Wyk, Oege de Moor, Kevin Backhouse, and Paul Kwiatkowski. Forwarding in Attribute Grammars for Modular Language Design. In *Compiler Construction (Proc. CC '02)*, volume 2304 of *Lecture Notes in Computer Science*, pages 128–142. Springer, 2002.

[WGS04] Ingo Wald, Johannes Günther, and Philipp Slusallek. Balancing Considered Harmful – Faster Photon Mapping using the Voxel Volume Heuristic –. In *Proc. Eurographics '04*, pages 595–604. Blackwell Publishing, 2004.

[WKSB07] Eric Van Wyk, Lijesh Krishnan, August Schwerdfeger, and Derek Bodin. Attribute Grammar-based Language Extensions for Java. In *ECOOP 2007 – Object-Oriented Programming*, volume 4609 of *Lecture Notes in Computer Science*, pages 575–599. Springer, 2007.

[WL91] Michael E. Wolf and Monica S. Lam. A Data Locality Optimizing Algorithm. In *Proc. PLDI '91*, pages 30–44. ACM, 1991.

[Won00] David Wonnacott. Using Time Skewing to Eliminate Idle Time due to Memory Bandwidth and Network Limitations. In *Proc. IPDPS '00*, pages 171–180. ACM, 2000.

[WWS+12] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-Optimizing AST Interpreters. In *Proc. DLS '12*, pages 73–82. ACM, 2012.

[WWW+13] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Duboscq Gilles, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to Rule Them All. In *Proc. Onward! '13*, pages 187–204. ACM, 2013.

[XKH04] Dana N. Xu, Siau-Cheng Khoo, and Zhenjiang Hu. PType System: A Featherweight Parallelizability Detector. In *Programming Languages and Systems (Proc. APLAS '04)*, volume 3302 of *Lecture Notes in Computer Science*, pages 197–212, 2004.

[YFRS13] Tomofumi Yuki, Paul Feautrier, Sanjay Rajopadhye, and Vijay Saraswat. Array Dataflow Analysis for Polyhedral X10 Programs. In *Proc. PPoPP '13*, pages 23–34. ACM, 2013.

[YKK+11] Kazuki Yoshizoe, Akihiro Kishimoto, Tomoyuki Kaneko, Haruhiro Yoshimoto, and Yutaka Ishikawa. Scalable Distributed Monte-Carlo Tree Search. In *Proc. SoCS '11*, pages 180–187. AAAI, 2011.

[YM06] Sung-Eui Yoon and Dinesh Manocha. Cache-Efficient Layouts of Bounding Volume Hierarchies. In *Proc. Eurographics '06*, pages 507–516. Blackwell Publishing, 2006.

[ZCD+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proc. NSDI '12*, pages 15–28. USENIX, 2012.

[ZLBF14] Wei Zhang, Per Larsen, Stefan Brunthaler, and Michael Franz. Accelerating Iterators in Optimizing AST Interpreters. In *Proc. OOPSLA '14*, pages 727–743. ACM, 2014.

# Publication List

1. Shigeyuki Sato and Akimasa Morihata.
   Syntax-Directed Divide-and-Conquer Data-Flow Analysis.
   In *Programming Languages and Systems — 12th Asian Symposium, APLAS 2014, Singapore, November 17–19, 2014, Proceedings*, *Lecture Notes in Computer Science*, Vol. 8858, pp. 392–407, Springer, 2014.
   Corresponding to Chapter 7.

2. Shigeyuki Sato and Kiminori Matsuzaki.
   A Generic Implementation of Tree Skeletons.
   In *Proc. 7th International Symposium on High-Level Parallel Programming and Applications*, pp. 51–72, Amsterdam, July 3–4, 2014.
   Accepted to International Journal of Parallel Programming.
   Corresponding to Chapter 3.

3. Shigeyuki Sato and Kiminori Matsuzaki.
   An Operator Generator for Skeletal Programming on Trees.
   *IPSJ Trans. PRO*, 6(4):38–49, 2013.
   In Japanese.
   Corresponding to Chapter 4.