Available online at www.sciencedirect.com
ScienceDirect
Parallel Computing 32 (2006) 590–603
PARALLEL COMPUTING
www.elsevier.com/locate/parco

Parallel skeletons for manipulating general trees

Kiminori Matsuzaki *, Zhenjiang Hu, Masato Takeichi

Graduate School of Information Science and Technology, The University of Tokyo, Tokyo 113-8656, Japan

Received 31 December 2005; received in revised form 20 May 2006; accepted 6 June 2006
Available online 4 August 2006

Abstract

Trees are important datatypes that are often used in representing structured data such as XML. Though trees are widely used in sequential programming, it is hard to write efficient parallel programs manipulating trees, because of their irregular and ill-balanced structures. In this paper, we propose a solution based on the skeletal approach. We formalize a set of skeletons (abstracted computational patterns) for rose trees (general trees of arbitrary shapes) based on the theory of Constructive Algorithmics. Our skeletons for rose trees are extensions of those proposed for lists and binary trees. We show that we can implement the skeletons efficiently in parallel, by combining the parallel binary-tree skeletons for which efficient parallel implementations are already known. As far as we are aware, we are the first who have formalized and implemented a set of simple but expressive parallel skeletons for rose trees.

Keywords: Skeletal parallelism; Tree skeletons; Rose trees; Constructive algorithmics

1. Introduction

Trees are important datatypes that are often used in representing structured data such as XML. In recent years, the growth of computational power enables us to store huge data in the form of trees. This calls for methods and systems of manipulating huge trees efficiently, where parallel computing may potentially be a solution. Though hardware environments for parallel computing are getting widely available (e.g., PC clusters), parallel programming is still considered to be a hard task, especially for trees because of their ill-balanced and irregular structures.

To resolve this problem, we adopt a novel paradigm of parallel programming called *skeletal parallelism*,[1] which was first proposed by Cole [1] and well discussed in [2]. In the skeletal parallelism, users build parallel programs by combining ready-made components called *parallel skeletons*. These parallel skeletons

* Corresponding author. Tel.: +81 3 5841 7412; fax: +81 3 5841 8607.
E-mail addresses: kmatsu@ipl.t.u-tokyo.ac.jp (K. Matsuzaki), hu@mist.i.u-tokyo.ac.jp (Z. Hu), takeichi@mist.i.u-tokyo.ac.jp (M. Takeichi).

[1] See http://homepages.inf.ed.ac.uk/mic/Skeletons/.

0167-8191/$ - see front matter
doi:10.1016/j.parco.2006.06.002

provide parallelizable computational patterns in a concise way and conceal the complicated parallel implementations from users. The skeletal parallelism has several advantages: the two most important ones are that users can build parallel programs as if they wrote sequential programs, and that the skeletal parallel programs are not only efficient but also architecture independent. There have been many studies on parallel skeletons for lists or arrays [3–9] and for binary trees [9–15], but there were few studies for general trees of arbitrary shapes.

This paper addresses parallel programming on *rose trees* [16], trees of arbitrary shapes. Many tree structures are straightforwardly formalized as rose trees, and many tree algorithms can be specified on these rose trees. We start by formalizing seven basic computational patterns over rose trees (rose-tree skeletons) based on the theory of Constructive Algorithmics [17–19]. Constructive Algorithmics was originally proposed for systematic development of sequential algorithms, and has also been applied to the specification of parallel skeletons for lists [5–9], matrices [20], and binary trees [9–11]. Our rose-tree skeletons are straightforward extensions of binary-tree skeletons where two new skeletons are added to describe computational patterns among siblings. We then show rose-tree skeletons can be implemented efficiently in parallel. We represent rose trees in the form of binary trees, and provide a mapping from the computation on rose-tree skeletons to that on binary trees specified with parallel binary-tree skeletons. Since the binary-tree skeletons can be implemented efficiently in parallel [11–14], our rose-tree skeletons can also be implemented in parallel, and thus we call them *parallel rose-tree skeletons*. We have implemented the rose-tree skeletons in C++ and MPI as wrapper functions of the binary-tree skeletons on our parallel skeleton library, and have obtained nice experimental results.

The rest of the paper is organized as follows. In Section 2, we briefly introduce the notations and review the parallel binary-tree skeletons. In Section 3, we formalize seven rose-tree skeletons with discussions about their expressiveness, and show their efficient parallel implementation in Section 4. We report experimental results in Section 5. Finally we discuss related work in Section 6, and make conclusion remarks in Section 7.

2. Preliminaries

In this section, after introducing important notational conventions used in this paper, we review parallel binary-tree skeletons [10,11].

2.1. Notations

In this paper, we borrow the notation of Haskell [21,22]. In the following, we briefly review important notations and define data structures. Roughly speaking, the definitions in Haskell in this paper can be read as mathematical function definition except for the function applications denoted by spaces.

2.1.1. Functions and operators

Function application is denoted by a space and the argument may be written without brackets. Thus $f\,a$ means $f(a)$. Functions are curried, and the function application associates to the left. Thus $f\,a\,b$ means $(f\,a)\,b$. The function application binds stronger than any other operator, so $f\,a \oplus b$ means $(f\,a) \oplus b$, but does not $f\,(a \oplus b)$. Function composition is denoted by an infix operator $\circ$. By definition, we have $(f \circ g)\,a = f\,(g\,a)$. Function composition is associative and its unit is the identity function denoted by *id*.

Infix binary operators will be denoted by $\oplus$, $\otimes$, etc., and their units are written as $\iota_{\oplus}$, $\iota_{\otimes}$, respectively. These operators can be sectioned and be treated as functions, i.e. $a \oplus b = (a \oplus)\,b = (\oplus b)\,a = (\oplus)\,a\,b$ holds.

In deriving parallel programs, algebraic rules on operators such as associativity or distributivity play important roles. We introduce the following generalized rule of distributivity defined as a closure property.

Definition 1 (*Extended distributivity* [23]). Let $\otimes$ be an associative operator. The operator $\otimes$ is said to be *extended-distributive* over operator $\oplus$, if there existfunctions p_1, p_2, and p_3 such that for any a, b, c, a', b', and c', the following equation holds:

$$(\lambda x.a \oplus (b \otimes x \otimes c)) \circ (\lambda x.a' \oplus (b' \otimes x \otimes c')) = \lambda x.A \oplus (B \otimes x \otimes C)$$

where

$$A = p_1(a, b, c, a', b', c'), \quad B = p_2(a, b, c, a', b', c'), \quad \text{and} \quad C = p_3(a, b, c, a', b', c').$$

We call functions p_1, p_2, and p_3 characteristic functions.

In fact, many pairs of operators satisfy this extended distributivity. For example, let $\oplus$ be an associative operator, then it is also extended-distributive over $\oplus$ itself. If operators $\oplus$ and $\otimes$ are associative operators and the operator $\otimes$ distributes over operator $\oplus$, then of course the operator $\otimes$ is extended-distributive over $\oplus$. Several other pairs of operators satisfy this property even if distributivity does not hold on them [23].

2.1.2. Lists and list comprehension

Cons lists (or simply lists) are finite sequences of elements of the same type. A list is constructed either by an empty list (*Nil*) or by adding an element to a list (*Cons*). The datatype of a list whose elements are of type α is defined as follows:

data *List* $\alpha = Nil | Cons\ \alpha\ (List\ \alpha)$.

We use abbreviations: $[\alpha]$ for *List* α, [ ] for *Nil*, and $(a{:}as)$ for (*Cons a as*).

List comprehension is a syntax sugar for generation of lists. Expression $[1..\#ts]$ generates a list of increasing integers from one to the number of elements in *ts*, and list comprehension $[f\ t_i | i \in [1..\#ts]]$ generates a list by applying function *f* to each element in *ts*. In this paper, we denote t_i for the *i*-th element of list *ts*, and we use similar notations for other lists too.

We introduce a notation for consumption of lists. Let $\oplus$ be an associative operator with its unit $\iota_\oplus$, then $\sum_\oplus$ denotes the reduction of a list with the operator $\oplus$. Informally, the $\sum_\oplus$ is defined as follows:

$$\sum_\oplus [\,] = \iota_\oplus$$
$$\sum_\oplus [a_1, a_2, \ldots, a_n] = a_1 \oplus a_2 \oplus \cdots \oplus a_n$$

We introduce two functions called *scans* or *prefix-sums*. The scan operation on lists, *scan*, takes an associative operator and a list, and accumulates values with the operator from left to right. Function *scan'* is a reversed scan operation. Informally, these two functions are defined as follows:

$$scan(\oplus)[a_1, a_2, \ldots, a_n] = [\iota_\oplus, a_1, \ldots, a_1 \oplus \cdots \oplus a_{n-1}]$$
$$scan'(\oplus)[a_1, a_2, \ldots, a_n] = [a_2 \oplus \cdots \oplus a_n, a_3 \oplus \cdots \oplus a_n, \ldots, a_n, \iota_\oplus]$$

2.1.3. Binary trees

Binary trees are trees whose internal nodes have exactly two children. The datatype of binary trees whose internal nodes have values of type α and leaves have values of type β is defined as follows:

data *BTree* $\alpha\ \beta = Leaf\ \alpha | Node\ \beta\ (BTree\ \alpha\ \beta)\ (BTree\ \alpha\ \beta)$.

We introduce two functions for manipulating binary trees. Function $root_b$ returns the value of the root node, and function $setroot_b$ takes a binary tree and a value, and replaces the value of the root node with the input value. Note that we use _ to denote a *do not care* value.

$$root_b\ (Leaf\ n) = n \quad setroot_b(Leaf\ n)a = Leaf\ a$$
$$root_b\ (Node\ n\ \_\ \_) = n \quad setroot_b\ (Node\ n\ l\ r)\ a = Node\ a\ l\ r$$

2.1.4. Rose trees

Rose trees (a term coined by Meertens [16]) are trees whose internal nodes have an arbitrary number of children. In this paper we assume all the nodes in a rose tree have values of the same type. The datatype of rose trees whose nodes have the values of type α is defined as follows using lists:

data *RTree* $\alpha = RNode\ \alpha\ [RTree\ \alpha]$.

Similar to binary trees, we introduce two functions for manipulating rose trees. Function $root_r$ returns the value of the root node, and function $setroot_r$ replaces the value of the root node with the input value.

$$root_r\ (RNode\ a\ ts) = a \qquad setroot_r\ (RNode\ a\ ts)\ b = RNode\ b\ ts$$

2.2. Basic binary-tree skeletons

The basic parallel binary-tree skeletons [10,11] are the following five higher-order functions, which are categorized as follows:

- Nodewise computations: *map* and *zipwith*
- Bottom–up computations: *reduce* and *upwards accumulate*
- Top–down computation: *downwards accumulate*

The definition of the parallel binary-tree skeletons is given in Fig. 1. In this paper, we denote the parallel skeletons for binary trees in the sans-serif font with a suffix b.

$$
\begin{aligned}
&\mathsf{map}_b :: (\alpha \to \gamma) \to (\beta \to \delta) \to BTree\ \alpha\ \beta \to BTree\ \gamma\ \delta\\
&\mathsf{map}_b\ k_L\ k_N\ (Leaf\ n) \quad = Leaf\ (k_L\ n)\\
&\mathsf{map}_b\ k_L\ k_N\ (Node\ n\ l\ r) = Node\ (k_N\ n)\ (\mathsf{map}_b\ k_L\ k_N\ l)\ (\mathsf{map}_b\ k_L\ k_N\ r)\\
&\mathsf{zipwith}_b :: (\alpha \to \alpha' \to \gamma) \to (\beta \to \beta' \to \delta) \to BTree\ \alpha\ \beta \to BTree\ \alpha'\ \beta'\\
&\qquad \to BTree\ \gamma\ \delta\\
&\mathsf{zipwith}_b\ k_L\ k_N\ (Leaf\ n)\ (Leaf\ n') = Leaf\ (k_L\ n\ n')\\
&\mathsf{zipwith}_b\ k_L\ k_N\ (Node\ n\ l\ r)\ (Node\ n'\ l'\ r')\\
&\quad = Node\ (k_N\ n\ n')\ (\mathsf{zipwith}_b\ k_L\ k_N\ l\ l')\ (\mathsf{zipwith}_b\ k_L\ k_N\ r\ r')\\
&\mathsf{reduce}_b :: (\beta \to \alpha \to \alpha \to \alpha) \to BTree\ \alpha\ \beta \to \alpha\\
&\mathsf{reduce}_b\ k\ (Leaf\ n) \quad = n\\
&\mathsf{reduce}_b\ k\ (Node\ n\ l\ r) = k\ n\ (\mathsf{reduce}_b\ k\ l)\ (\mathsf{reduce}_b\ k\ r)\\
&\mathsf{uAcc}_b :: (\beta \to \alpha \to \alpha \to \alpha) \to BTree\ \alpha\ \beta \to BTree\ \alpha\ \alpha\\
&\mathsf{uAcc}_b\ k\ (Leaf\ n) \quad = Leaf\ n\\
&\mathsf{uAcc}_b\ k\ (Node\ n\ l\ r) = \mathbf{let}\ l' = \mathsf{uAcc}_b\ k\ l\\
&\qquad\qquad\qquad\qquad\quad r' = \mathsf{uAcc}_b\ k\ r\\
&\qquad\qquad\qquad\quad \mathbf{in}\ \ Node\ (k\ n\ (root_b\ l')\ (root_b\ r'))\ l'\ r'\\
&\mathsf{dAcc}_b :: (\gamma \to \gamma \to \gamma) \to (\beta \to \gamma) \to (\beta \to \gamma) \to BTree\ \alpha\ \beta \to BTree\ \gamma\\
&\mathsf{dAcc}_b\ (\oplus)\ g_l\ g_r\ t \quad = \mathsf{dAcc}'_b\ (\oplus)\ g_l\ g_r\ \iota_\oplus\ t\\
&\mathsf{dAcc}'_b\ (\oplus)\ g_l\ g_r\ c\ (Leaf\ n) \quad = Leaf\ c\\
&\mathsf{dAcc}'_b\ (\oplus)\ g_l\ g_r\ c\ (Node\ n\ l\ r) = Node\ c\ (\mathsf{dAcc}'_b\ (\oplus)\ g_l\ g_r\ (c \oplus g_l\ n)\ l)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (\mathsf{dAcc}'_b\ (\oplus)\ g_l\ g_r\ (c \oplus g_r\ n)\ r)\\
&\mathsf{getchl}_b :: \alpha \to BTree\ \beta\ \beta \to BTree\ \alpha\ \beta\\
&\mathsf{getchl}_b\ c\ (Leaf\ n) \quad = Leaf\ c\\
&\mathsf{getchl}_b\ c\ (Node\ n\ l\ r) = Node\ (root_b\ l)\ (\mathsf{getchl}_b\ c\ l)\ (\mathsf{getchl}_b\ c\ r)\\
&\mathsf{getchr}_b :: \alpha \to BTree\ \beta\ \beta \to BTree\ \alpha\ \beta\\
&\mathsf{getchr}_b\ c\ (Leaf\ n) \quad = Leaf\ c\\
&\mathsf{getchr}_b\ c\ (Node\ n\ l\ r) = Node\ (root_b\ r)\ (\mathsf{getchr}_b\ c\ l)\ (\mathsf{getchr}_b\ c\ r)
\end{aligned}
$$

Fig. 1. Definition of binary-tree skeletons.

Efficient implementations of these parallel skeletons have been studied [12–14] based on the tree contraction algorithms [24,25]. The tree contraction algorithms are important parallel algorithms for manipulating binary trees of arbitrary shapes efficiently.

The parallel skeleton map_b takes two functions k_L and k_N and a binary tree, and applies k_L to each leaf and k_N to each internal node. The parallel skeleton $\mathsf{zipwith}_b$ takes two functions k_L and k_N and two binary trees of the same shape, and zips the trees up by applying k_L to each pair of leaves and k_N to each pair of internal nodes.

The parallel skeleton reduce_b takes a function k and a binary tree, and collapses the tree into a value by applying the function k in a bottom–up manner. The parallel skeleton uAcc_b takes a function k and a binary tree, and computes $\mathsf{reduce}_b\, k$ for each subtree. In other words, this skeleton applies the function k in a bottom–up manner, and returns a tree whose values are the results of bottom–up reduction. To guarantee existence of an efficient parallel implementation of the reduce_b and uAcc_b skeletons, we require the existence of functions ϕ, ψ_L, ψ_R, and G, satisfying the following equations:

$$k\ n\ x\ y = G\ (\phi\ n)\ x\ y$$
$$G\ n\ l\ (G\ n'\ x\ y) = G\ (\psi_L\ n\ l\ n')\ x\ y$$
$$G\ n\ (G\ n'\ x\ y)\ r = G\ (\psi_R\ n\ r\ n')\ x\ y$$

We denote the function k satisfying the condition above as $k = \langle \phi, \psi_L, \psi_R, G \rangle$. When these functions exist we can implement the reduce_b and uAcc_b skeletons based on the tree contraction algorithms [10,14].

The parallel skeleton dAcc_b takes an associative operator $\oplus$, two functions g_l and g_r, and a binary tree. This skeleton computes in a top–own manner by updating accumulative parameter c, whose initial value is the unit of the operator, $\iota_\oplus$. The accumulative parameter is updated with $\oplus$ and g_l for the left child, and with $\oplus$ and g_r for the right child. The condition for efficient parallel computation is the associativity of the operator $\oplus$.

We briefly remark on the parallel computation cost of the skeletons. In the discussion of parallel computation cost, N indicates the number of nodes in a tree, and P the number of processors. The parallel computation cost of $\mathsf{map}_b\, k_L\, k_N$ and $\mathsf{zipwith}_b\, k_L\, k_N$ is $\mathrm{O}(N/P)$ if the functions k_L and k_N are computed in constant time. When there exist constant-cost functions satisfying the condition of $\mathsf{reduce}_b\, k$ and $\mathsf{uAcc}_b\, k$, then these skeletons can be computed in $\mathrm{O}(N/P + \log P)$ parallel time using the tree contraction techniques. The parallel computation cost of $\mathsf{dAcc}_b\ (\oplus)\ g_l\, g_r$ is also $\mathrm{O}(N/P + \log P)$, if the operator $\oplus$ and the functions g_l and g_r are computed in constant time.

2.3. Specialized binary-tree skeletons

We define two communication skeletons for reasons of readability and efficiency. The parallel skeleton getchl_b takes a value and a binary tree, and puts the left child's value for each internal node and the input value for each leaf. In other words, this skeleton shifts each left child's value to its parent. The parallel skeleton getchr_b is symmetric to the getchl_b skeleton: this skeleton takes a value and a binary tree, and put the right child's value for each internal node and the input value for each leaf.

In fact we can express them with the map_b and uAcc_b skeletons. We can furthermore implement the getchl_b and getchr_b skeletons to run in $\mathrm{O}(N/P)$ parallel time, since the dependency in these skeletons is local.

3. Rose-tree skeletons

In this section, we formalize computational patterns on rose trees based on the theory of Constructive Algorithmics [17–19], and illustrate how we can develop programs with these skeletons.

3.1. Specification of rose-tree skeletons

The key idea of Constructive Algorithmics is that the computation structure of algorithms should be derivable from the data structures the algorithms manipulate. We have defined rose trees as trees whose internal node has a list of children, and thus, we specify computational patterns on rose trees by extending those

on binary trees with computations on lists. We formalize seven skeletons on rose trees, which are categorized into the following four groups:

- Nodewise computations: *map* and *zipwith*,
- Bottom–up computations: *reduce* and *upwards accumulate*,
- Top–down computation: *downwards accumulate*, and
- Intra-siblings computations: *rightwards accumulate* and *leftwards accumulate*.

We shall denote the rose-tree skeletons in the sans-serif font with a suffix r. In this paper, we give an intuitive specification of them using list comprehensions (Fig. 2). Their formal definition specified as mutual recursive functions is summarized in the technical report [26].

We define nodewise computations similar to those on binary trees. Rose-tree skeleton map_r takes a function k and a rose tree, and applies the function to each node. Rose-tree skeleton $\mathsf{zipwith}_r$ takes a function k

$$
\begin{aligned}
&\mathsf{map}_r :: (\alpha \to \beta) \to \mathit{RTree}\ \alpha \to \mathit{RTree}\ \beta\\
&\mathsf{map}_r\ k\ (\mathit{RNode}\ a\ ts) = \mathit{RNode}\ (k\ a)\ [\mathsf{map}_r\ k\ t_i \mid i \in [1..\#ts]]\\
&\\
&\mathsf{zipwith}_r :: (\alpha \to \alpha' \to \beta) \to \mathit{RTree}\ \alpha \to \mathit{RTree}\ \alpha' \to \mathit{RTree}\ \beta\\
&\mathsf{zipwith}_r\ k\ (\mathit{RNode}\ a\ ts)\ (\mathit{RNode}\ a'\ ts')\\
&\qquad = \mathit{RNode}\ (k\ a\ a')\ [\mathsf{zipwith}_r\ k\ t_i\ t'_i \mid i \in [1..\#ts]]\\
&\\
&\mathsf{reduce}_r :: (\alpha \to \beta \to \beta) \to (\beta \to \beta \to \beta) \to \mathit{RTree}\ \alpha \to \beta\\
&\mathsf{reduce}_r\ (\oplus)\ (\otimes)\ (\mathit{RNode}\ a\ ts) = a \oplus \textstyle\sum_{\otimes}[\mathsf{reduce}_r\ (\oplus)\ (\otimes)\ t_i \mid i \in [1..\#ts]]\\
&\\
&\mathsf{uAcc}_r :: (\alpha \to \beta \to \beta) \to (\beta \to \beta \to \beta) \to \mathit{RTree}\ \alpha \to \mathit{RTree}\ \beta\\
&\mathsf{uAcc}_r\ (\oplus)\ (\otimes)\ (\mathit{RNode}\ a\ ts) = \mathit{RNode}\ (\mathsf{reduce}_r\ (\oplus)\ (\otimes)\ (\mathit{RNode}\ a\ ts))\\
&\qquad\qquad [\mathsf{uAcc}_r\ (\oplus)\ (\otimes)\ t_i \mid i \in [1..\#ts]]\\
&\\
&\mathsf{rAcc}_r :: (\alpha \to \alpha \to \alpha) \to \mathit{RTree}\ \alpha \to \mathit{RTree}\ \alpha\\
&\mathsf{rAcc}_r\ (\oplus)\ (\mathit{RNode}\ a\ ts)\\
&\qquad = \mathbf{let}\ rs = \mathit{scan}\ (\oplus)\ [\mathit{root}_r\ t_i \mid i \in [1..\#ts]]\\
&\qquad\quad \mathbf{in}\ \ \mathit{RNode}\ \iota_{\oplus}\ [\mathit{setroot}_r\ (\mathsf{rAcc}_r\ (\oplus)\ t_i)\ r_i \mid i \in [1..\#ts]]\\
&\\
&\mathsf{lAcc}_r :: (\alpha \to \alpha \to \alpha) \to \mathit{RTree}\ \alpha \to \mathit{RTree}\ \alpha\\
&\mathsf{lAcc}_r\ (\oplus)\ (\mathit{RNode}\ a\ ts)\\
&\qquad = \mathbf{let}\ rs = \mathit{scan}'\ (\oplus)\ [\mathit{root}_r\ t_i \mid i \in [1..\#ts]]\\
&\qquad\quad \mathbf{in}\ \ \mathit{RNode}\ \iota_{\oplus}\ [\mathit{setroot}_r\ (\mathsf{lAcc}_r\ (\oplus)\ t_i)\ r_i \mid i \in [1..\#ts]]\\
&\\
&\mathsf{dAcc}_r :: (\alpha \to \alpha \to \alpha) \to \mathit{RTree}\ \alpha \to \mathit{RTree}\ \alpha\\
&\mathsf{dAcc}_r\ (\oplus)\ t \qquad\qquad\quad = \mathsf{dAcc}'_r\ (\oplus)\ \iota_{\oplus}\ t\\
&\mathsf{dAcc}'_r\ (\oplus)\ c\ (\mathit{RNode}\ a\ ts) = \mathit{RNode}\ c\ [\mathsf{dAcc}'_r\ (\oplus)\ (c \oplus a)\ t_i \mid i \in [1..\#ts]]\\
&\\
&\mathsf{rAcc}_r :: (\alpha \to \alpha \to \alpha) \to \mathit{RTree}\ \alpha \to \mathit{RTree}\ \alpha\\
&\mathsf{rAcc}_r\ (\oplus)\ (\mathit{RNode}\ a\ ts)\\
&\qquad = \mathbf{let}\ rs = \mathit{scan}\ (\oplus)\ [\mathit{root}_r\ t_i \mid i \in [1..\#ts]]\\
&\qquad\quad \mathbf{in}\ \ \mathit{RNode}\ \iota_{\oplus}\ [\mathit{setroot}_r\ (\mathsf{rAcc}_r\ (\oplus)\ t_i)\ r_i \mid i \in [1..\#ts]]\\
&\\
&\mathsf{lAcc}_r :: (\alpha \to \alpha \to \alpha) \to \mathit{RTree}\ \alpha \to \mathit{RTree}\ \alpha\\
&\mathsf{lAcc}_r\ (\oplus)\ (\mathit{RNode}\ a\ ts)\\
&\qquad = \mathbf{let}\ rs = \mathit{scan}'\ (\oplus)\ [\mathit{root}_r\ t_i \mid i \in [1..\#ts]]\\
&\qquad\quad \mathbf{in}\ \ \mathit{RNode}\ \iota_{\oplus}\ [\mathit{setroot}_r\ (\mathsf{lAcc}_r\ (\oplus)\ t_i)\ r_i \mid i \in [1..\#ts]]
\end{aligned}
$$

Fig. 2. Definition of rose-tree skeleton.

and two rose trees of the same shape, and zips them up by applying the function to each pair of corresponding nodes.

Since rose trees are defined with lists, we specify rose-tree skeleton reduce_r with two operators: one for folding the list of children, and the other for merging the result of children with their parent. Thus, rose-tree skeleton reduce_r takes two operators and a rose tree, and collapses the tree into a value in a bottom–up manner. By definition, the operator $\otimes$ should be associative. For efficient parallel implementations, we furthermore require some conditions on the two operators. We provide the following two sufficient conditions.

- The operators $\oplus$ and $\otimes$ form an algebraic semi-ring, that is, both $\oplus$ and $\otimes$ are associative with their units, and $\oplus$ is distributive over $\otimes$.
- The operator $\otimes$ is extended-distributive over $\oplus$.

This computational pattern is also called homomorphism on rose trees, and was discussed in our previous paper [23] under the second condition.

We then define another bottom–up computational pattern. Rose-tree skeleton uAcc_r (upwards accumulate) takes two operators and a rose tree, and applies the reduce_r skeleton for each subtree. Thus, this skeleton returns a rose-tree of the same shape as the input tree. For efficient implementations of the uAcc_r skeleton, we require the same conditions as the reduce_r skeleton.

Rose-tree skeleton dAcc_r (downwards accumulate) is a top–down computational pattern. By Constructive Algorithmics, we consider that the dAcc_r skeleton applies list-consuming operation $\sum_{\oplus}$ to each path from the root. Similar to the dAcc_b skeleton on binary trees, we define the dAcc_r skeleton with an accumulative parameter c as follows. Note that we omit the function applied to each node, since we can use the map_r skeleton instead.

The five rose-tree skeletons above are extensions of those on binary trees. We add two skeletons that are specific to rose trees by formalizing computations among siblings. Rose-tree skeleton rAcc_r (rightwards accumulate) takes an associative operator $\oplus$ and a rose tree, and applies the list-consuming operation to the list of the left-siblings for each node. We specify this skeleton to apply the scan operation to each list of siblings. Rose-tree skeleton lAcc_r (leftwards accumulate) is symmetric to the rAcc_r skeleton. This skeleton applies the reversed scan operation to each list of siblings. Note that r_i indicates the i-th element of the rs in the definition in Fig. 2.

3.2. Example: preorder numbering problem

To see how we can use these rose-tree skeletons for manipulating general trees, we develop a skeletal program for the preorder numbering problem on rose trees. For a given rose tree, we want to assign a number for each node in the order of the preorder traversal. In the preorder numbering, the number of a leftmost child is larger than that of its parent by one, and the number of another child is larger than its left sibling by the number of nodes in the left-sibling's subtree. We can give a recursive algorithm for the preorder numbering problem as follows with auxiliary function *size* that counts the number of nodes in a tree.

$$
\begin{aligned}
&\mathit{pre}\ t && = \mathit{pre}'\ 0\ t \\
&\mathit{pre}'\ c\ (\mathit{RNode}\ a\ ts) && = \mathit{RNode}\ c\ [\mathit{pre}'\ (c + 1 + l_i)\ t_i | i \in [1..\#ts]] \\
& && \quad \textbf{where}\ l_i = \sum_{+} [\mathit{size}\ t_j | j \in [1..i-1]] \\
&\mathit{size}\ (\mathit{RNode}\ a\ ts) && = 1 + \sum_{+} [\mathit{size}\ t_i | i \in [1..\#ts]]
\end{aligned}
$$

We can compute the results of *size* by a bottom–up computation using the uAcc_r and map_r skeletons. We then apply the rAcc_r skeleton to accumulate the results from left to right. Finally, we compute the results of preorder numbering by a top–down computation with the map_r, dAcc_r, and $\mathsf{zipwith}_r$ skeletons. Therefore, we can give a skeletal program for the preorder numbering problem as follows. For the detailed derivation of the skeletal program, see our technical report [26].

$$
\begin{aligned}
pre\ t = \textbf{let}\ & lt = \mathsf{rAcc}_r\ (+)\ (\mathsf{uAcc}_r\ (+)\ (+)\ (\mathsf{map}_r\ (\lambda x.1)\ t)) \\
& gt = setroot_r\ 0\ (\mathsf{map}_r\ (\lambda l.1 + l)\ lt) \\
& dt = \mathsf{dAcc}_r\ (+)\ gt \\
\textbf{in}\ & \mathsf{zipwith}_r\ (+)\ dt\ gt
\end{aligned}
$$

4. Parallelizing rose-tree skeletons with binary-tree skeletons

In this section, we show a parallel implementation for the rose-tree skeletons. The main idea is to represent rose trees by binary trees and to implement the rose-tree skeletons by the binary-tree skeletons. Since we can implement the binary-tree skeletons efficiently in parallel [12–14], we can implement the rose-tree skeletons efficiently as well.

4.1. Binary-tree representation of rose trees

Many researchers have studied the parallel manipulations of binary trees based on the parallel tree contraction algorithms [24,25,27,28], and we can implement the binary-tree skeletons based on the tree contraction algorithms [12,13]. To utilize these parallel implementations, we represent the rose trees in the forms of binary trees as shown in Fig. 3. This binary-tree representation is one of the widely-used representations [29], but for parallel programming this representation was rarely used. In this representation, every internal node comes from a node in the original rose tree, and all leaves are dummy nodes. The left child of a node in the binary tree is its leftmost child in the original rose tree, and the right child of a node in the binary tree is its next sibling in the rose tree. Let n be the number of nodes of the original rose tree, then the number of nodes of the binary tree turns out to be $2n + 1$. This guarantees the asymptotic cost when we utilize the parallel binary-tree skeletons on this binary-tree representation.

To formalize the binary-tree representation, we define functions $r2b$ that transforms a rose tree into its binary-tree representation, and $b2r$ that restores a rose tree from its binary-tree representation. The definition of these two functions is shown in Fig. 4. Note that $b2r \circ r2b = id$ holds.

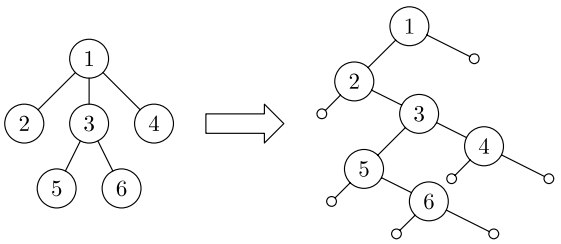

Fig. 3. The binary-tree representation of rose trees.

$$
\begin{aligned}
& r2b :: RTree\ \alpha \to BTree\ \_\ \alpha && b2r :: BTree\ \_\ \alpha \to RTree\ \alpha \\
& r2b\ t = r2b'\ t\ [\,] && b2r\ t = head\ (b2r'\ t) \\
& r2b'\ (RNode\ a\ ts)\ ss && b2r'\ (Node\ n\ l\ r) \\
& \quad = Node\ a\ (r2b''\ ts)\ (r2b''\ ss) && \quad = (RNode\ n\ (b2r'\ l)) : b2r'\ r \\
& r2b''\ [\,] = Leaf\ \_ && b2r'\ (Leaf\ n) \\
& r2b''\ (t : ts) = r2b'\ t\ ts && \quad = [\,]
\end{aligned}
$$

Fig. 4. Definition of functions $r2b$ and $b2r$.

4.2. Parallelization of rose-tree skeletons with binary-tree skeletons

We implement the rose-tree skeletons on the binary-tree representation using the parallel binary-tree skeletons. Generally speaking, our implementation of a rose-tree skeleton consists of three steps: (1) applying the function $r2b$ to transform a rose tree to a binary tree; (2) applying binary-tree skeletons to perform the computation of the skeleton; and (3) applying the function $b2r$ to restore the rose-tree structure if necessary.

Since every node in a rose tree is an internal node in the binary-tree representation, and there are no dependencies in the computation of the map_r skeleton, we can implement the map_r skeleton by simply using the map_b skeleton to apply the function to each internal node. Here, we do not care the function for leaves. Similarly, we can also implement the $\mathsf{zipwith}_r$ skeleton.

$$\mathsf{map}_r\ k = b2r \circ (\mathsf{map}_b\ \_k) \circ r2b$$
$$\mathsf{zipwith}_r\ k\ t\ t' = b2r\ (\mathsf{zipwith}_b\ \_k\ (r2b\ t)\ (r2b\ t'))$$

The computation of the reduce_r skeleton is also a bottom–up one on the binary-tree representation, and thus we can implement it with the map_b and reduce_b skeletons as follows. Note that we need not to apply the $b2r$ function since the reduce_r skeleton returns a value not a tree.

$$\mathsf{reduce}_r\ (\oplus) = (\mathsf{reduce}_b\ k) \circ (\mathsf{map}_b\ (\lambda x.\iota_\otimes)\ id) \circ r2b$$
$$\textbf{where}\ k\ n\ l\ r = (n \oplus l) \otimes r$$

For the parallel implementation, the function k above should satisfy the condition of the reduce_b skeleton. Though it is hard to give a general derivation method, we show that we can derive functions under the two sufficient conditions. We follow the derivation method in [10], where the main idea is to introduce a parametrized function closed under nested calls. Let $G[a]$ be a parametrized binary-function with parameter a, and we call it closed under nested calls if the following two equations hold.

$$G[n]\ l\ (G[rn]\ rl\ rr) = G[\psi_L\ n\ l\ rn]\ rl\ rr$$
$$G[n]\ (G[ln]\ ll\ lr)\ r = G[\psi_R\ n\ r\ ln]\ ll\ lr$$

In the following, we show the parametrized functions and the results of the parallel implementation of the reduce_r skeleton.

Firstly, let $\oplus$ and $\otimes$ form an algebraic semi-ring. In this case, we choose the parametrized function with three parameters a, b, and c as

$$G[(a,b,c)] = \lambda lr.(a \oplus l) \otimes (b \oplus r) \otimes c.$$

From this parametrized function we derive the functions ϕ, ψ_L, and ψ_R for the reduce_b skeleton, and we successfully obtain an equivalent definition of the reduce_r skeleton on the binary-tree representation as follows:

$$\mathsf{reduce}_r\ (\oplus)\ (\otimes) = (\mathsf{reduce}_b\ \langle \phi, \psi_L, \psi_R, G\rangle) \circ (\mathsf{map}_b\ (\lambda x.\iota_\otimes)\ id) \circ r2b$$
$$\textbf{where}\ \phi\ n = (n, \iota_\oplus, \iota_\otimes)$$
$$\psi_L\ (a,b,c)\ l'\ (a',b',c') = (b \oplus a', b \oplus b', (a \oplus l') \otimes (b \oplus c') \otimes c)$$
$$\psi_R\ (a,b,c)\ r'\ (a',b',c') = (a \oplus a', a \oplus b', (a \oplus c') \otimes (b \oplus r') \otimes c)$$
$$G\ (a,b,c)\ l\ r = (a \oplus l) \otimes (b \oplus r) \otimes c$$

Secondly, let the operator $\otimes$ be extended-distributive over $\oplus$ with characteristic functions p_1, p_2, and p_3. Here, we choose the parametrized function with four parameters a, b, c, and d as

$$G[(a,b,c,d)] = \lambda l\ r.a \oplus (b \otimes (c \oplus l) \otimes r \otimes d).$$

From this parametrized function we derive the functions ϕ, ψ_L, and ψ_R for the reduce_b skeleton, and we obtain the following parallel implementation of the reduce_r skeleton as follows:

$$\begin{aligned}
\mathsf{reduce}_r\ (\oplus)\ (\otimes) = {} & (\mathsf{reduce}_b\ [\phi, \psi_L, \psi_R, G]) \circ (\mathsf{map}_b(\lambda x.\iota_\otimes)\ id) \circ r2b \\
& \textbf{where}\ \phi\ n = (\iota_\oplus, \iota_\otimes, n, \iota_\otimes) \\
& \qquad \psi_L\ (a,b,c,d)\ l'\ (a',b',c',d') \\
& \qquad\quad = \textbf{let}\ tup = (a, b \otimes (c \oplus l'),\ d,\ a',\ b',\ d') \\
& \qquad\qquad \textbf{in}\ (p_1\ tup, p_2\ tup, c', p_3\ tup) \\
& \qquad \psi_R\ (a,b,c,d)\ r'\ (a',b',c',d') \\
& \qquad\quad = \textbf{let}\ tup' = (c, \iota_\otimes, \iota_\otimes, a', b', d') \\
& \qquad\qquad\quad tup = (a, b, r' \otimes d, p_1\ tup', p_2\ tup', p_3\ tup') \\
& \qquad\qquad \textbf{in}\ (p_1\ tup, p_2\ tup, c', p_3\ tup) \\
& \qquad G\ (a,b,c,d)\ l\ r = a \oplus (b \otimes (c \oplus l) \otimes r \otimes d)
\end{aligned}$$

Since the uAcc_r skeleton is similar to the reduce_r skeleton, first let us consider applying the map_b and uAcc_b skeletons with the same functions used in parallelizing the reduce_r skeleton.

$$\begin{aligned}
& b2r \circ (\mathsf{uAcc}_b\ k) \circ (\mathsf{map}_b\ (\lambda x.\iota_\otimes)\ id) \circ r2b \\
& \textbf{where}\ k\ n\ l\ r = (n \oplus l) \otimes r
\end{aligned}$$

Unfortunately, the results are not what we want for the uAcc_r skeleton. Since an internal node in the binary-tree representation has a right-child that was a sibling in the original rose tree, the result of the program above includes the siblings' values. To obtain the desired result, we need to compute $(n \oplus l')$ again for each internal node where n is the original node's value and l' is the left-child's value in the result of the uAcc_b skeleton above. We realize this computation by the getchl_b and $\mathsf{zipwith}_b$ skeletons.

Therefore, the equivalent definition of the uAcc_r skeleton under the binary-tree representation is given as follows. Note that the four functions for the parallel implementation of the uAcc_b skeleton, ϕ, ψ_L, ψ_R, and G, are the same as those derived for implementing the reduce_r skeleton.

$$\begin{aligned}
\mathsf{uAcc}_r\ (\oplus)\ (\otimes)\ t = {} & \textbf{let}\ bt\ = r2b\ t \\
& \qquad\qquad bt' = \mathsf{uAcc}_b\ \langle \phi, \psi_L, \psi_R, G \rangle\ (\mathsf{map}_b\ (\lambda x.\iota_\otimes)\ id\ bt) \\
& \qquad\ \textbf{in}\ b2r\ (\mathsf{zipwith}_b\ _\ (\lambda n\ l'.n \oplus l')\ bt\ (\mathsf{getchl}_b\ _\ bt'))
\end{aligned}$$

The dAcc_r skeleton is a top–down computational pattern on the binary-tree representation. By introducing a recursive function with an accumulative parameter, we can specify the dAcc_r skeleton.

$$\begin{aligned}
\mathsf{dAcc}_r\ (\oplus) = {} & b2r \circ (f\ \iota_\oplus) \circ r2b \\
& \textbf{where}\ f\ c\ (Leaf\ _) = Leaf\ _ \\
& \qquad f\ c\ (Node\ n\ l\ r) = Node\ c\ (f\ (c \oplus n)\ l)\ (f\ c\ r)
\end{aligned}$$

Noting that we can compute the accumulative parameter for the right child as $c = c \oplus \iota_\oplus = c \oplus (\lambda x \cdot \iota_\oplus)\ n$, we can implement the function f with the dAcc_b skeleton. Therefore, a parallel implementation of the dAcc_r skeleton is as follows:

$$\mathsf{dAcc}_r\ (\oplus) = b2r \circ (\mathsf{dAcc}_b\ (\oplus)\ id\ (\lambda x.\iota_\oplus)) \circ r2b$$

The skeleton rAcc_r traverses the siblings from left to right on rose trees, which corresponds to a top–down traversal on binary trees. Thus, we implement the rAcc_r skeleton on the binary-tree representation with the following recursive function f:

$$\begin{aligned}
\mathsf{rAcc}_r\ (\oplus) = {} & b2r \circ (f\ \iota_\oplus) \circ r2b \\
& \textbf{where}\ f\ c\ (Leaf\ _) = Leaf\ _ \\
& \qquad f\ c\ (Node\ n\ l\ r) = Node\ c\ (f\ \iota_\oplus\ l)(f\ (c \oplus n)\ r)
\end{aligned}$$

Though this function f is a top–down computation on the binary-tree representation, in fact we cannot simply describe it with the dAcc_b skeleton. This is because we cannot find a function for the left-child, g_l, such that for any n, $\iota_\oplus = c \oplus g_l\ n$ holds. Therefore, we need to derive another suitable function with an associative operator.

To derive such an associative operator, we utilize the context preservation technique [4], which derives an associative operator from a parametrized function closed under function composition. Here, we choose a parametrized function defined with three parameters p, a, and b, as follows:

$$\lambda x.\textbf{if } p \textbf{ then } x \oplus a \textbf{ else } b.$$

Based on this parametrized function, we derive an associative operator $\otimes$ defined as follows.

$$\begin{aligned}
(True, p, a, b) \otimes (True, p', a', b') &= (True, p' \wedge p, a \oplus a',\ \textbf{if } p' \textbf{ then } b \oplus a' \textbf{ else } b') \\
(True, p, a, b) \otimes (False, _, _, _) &= (True, p, a, b) \\
(False, _, _, _) \otimes (flag, p, a, b) &= (flag, p, a, b)
\end{aligned}$$

Here, the unit of $\otimes$ is $\iota_{\otimes} = (False, True, \iota_{\oplus}, \_)$. We introduce the first value of the tuple to represent the left-unit of $\otimes$. Using this operator, we successfully derive a parallel implementation of the rAcc_r skeleton.

$$\begin{aligned}
\mathsf{rAcc}_r(\oplus) = b2r \circ\ & (\mathsf{map}_b\ _\ k) \circ (\mathsf{dAcc}_r\ (\otimes)\ g_l\ g_r) \circ r2b \\
\textbf{where } g_l\, x &= (True, False, _, \iota_{\oplus}) \\
g_r\, x &= (True, True, x, _) \\
k(_, p, a, b) &= \textbf{if } p \textbf{ then } a \textbf{ else } b
\end{aligned}$$

The skeleton lAcc_r traverses the siblings from right to left, which corresponds to a bottom–up traversal on the binary-tree representation. Therefore, an implementation of the lAcc_r skeleton on the binary-tree representation may have a call of the uAcc_b skeleton. We first consider the following composition of the uAcc_b and map_b skeletons.

$$\begin{aligned}
& b2r \circ (\mathsf{uAcc}_b\ k) \circ (\mathsf{map}_b\ (\lambda x.\iota_{\oplus})\ id) \circ r2b \\
& \textbf{where } k\ n\ l\ r = n \oplus r
\end{aligned}$$

The results of this computation are slightly different from what we want for the lAcc_r skeleton: the results should be shifted to the left by one on the rose tree. We resolve this problem by applying the getchr_b skeleton before restoring the rose-tree structure with the $b2r$ function.

For the parallel implementation, we next show the function k satisfies the condition of the uAcc_b skeleton. For the function k above, we introduce a parametrized function with three parameters p, a, and b as

$$G[(p, a, b)] = \lambda l\ r.\textbf{if } p \textbf{ then } a \oplus r \textbf{ else } b$$

Based on this function, we successfully obtain an parallel implementation of the lAcc_r skeleton as follows.

$$\begin{aligned}
\mathsf{lAcc}_r(\oplus) = b2r \circ (\mathsf{getchr}_b\ _) \circ (\mathsf{uAcc}_b\langle \phi, \psi_L, \psi_R, G\rangle) \circ (\mathsf{map}_b\ (\lambda x.\iota_{\oplus})\ id) \circ r2b \\
\textbf{where } \phi\ n = (True, n, _) \\
\psi_L(p, a, b)\ l'\ (p', a', b') = (p \wedge p', a \oplus a',\ \textbf{if } p \textbf{ then } a \oplus b' \textbf{ else } b) \\
\psi_R\ (p, a, b)\ r'\ (p', a', b') = (False, _,\ \textbf{if } p \textbf{ then } a \oplus r' \textbf{ else } b) \\
G\ (p, a, b)\ l\ r = \textbf{if } p \textbf{ then } a \oplus r \textbf{ else } b
\end{aligned}$$

Now we briefly discuss the efficiency of our parallel implementation of the skeletons. We used two functions $r2b$ and $b2r$ for specifying the computation on the binary-tree representation. However, we can remove these two functions away if two rose-tree skeletons are successively called. Thus we give the parallel cost of the rose-tree skeletons without these two functions. The parallel cost of the map_r and $\mathsf{zipwith}_r$ skeletons are $O(N/P)$, and the parallel cost of the other five skeletons are $O(N/P + \log P)$, where N denotes the number of nodes of the rose trees, and P the number of processors. Note that the implementation of the rose-tree skeletons is cost optimal.

We may develop more involved implementations of the rose-tree skeletons by removing unnecessary intermediate structures and optimizing sequential parts. To develop such an implementation is our future work.

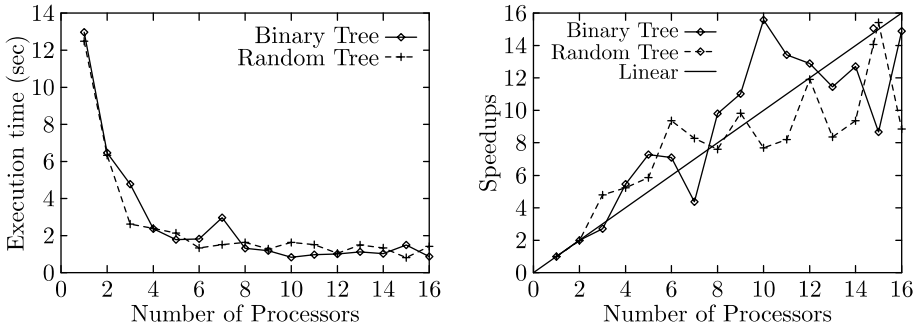

Fig. 5. Experimental results.

We summarize this section with the following theorem.

Theorem 1. *The seven parallel skeletons for rose trees defined in Fig. 2 can be implemented in parallel based on the binary-tree representation with the parallel binary-tree skeletons.*

Proof. The correctness of the implementation of the rose-tree skeletons is almost self-evident from the derivation so far. For more detailed derivation, see our technical report [26]. □

5. Experiments

We have implemented the rose-tree skeletons as wrapper functions of the binary-tree skeletons in our skeleton library, SkeTo [30,31].[2] To see the efficiency of the skeletons, we have made experiments with the preorder numbering problem in Section 3.

The environment is our PC cluster that consists of sixteen uniform PCs connected with Gigabit Ethernet. Each PC has a CPU of Pentium4 3.0 GHz (Hyper Threading ON) and 1GB memory. The OS, the C++ compiler, and the MPI library are Linux 2.6.8, gcc 2.95, and mpich 1.2.6, respectively. We executed the skeletal program with varying number of CPUs used for two trees of $2^{22} - 1$ nodes (almost four million). The first tree is a perfect binary tree, and the second tree is a randomly generated tree whose height is seven (this height comes from the average height of XML trees [32]).

The experimental results are shown in Fig. 5. The execution times do not include initial transformation/distribution and final gathering. The execution times for the binary tree are 13.0 s with one processor and 0.87 s with sixteen processors, and the execution times for the random tree are 12.5 s with one processor and 1.41 s with sixteen processors. For both data, the skeletal program scaled well, in particular, the speedup is 14.9 with sixteen processors for the binary tree. In some points the execution times are a little worse (e.g., with seven processors for the binary tree), but this is due to the ill-balanced partitioning of trees (in terms of the size of partitioned segments) on our implementation of the binary-tree skeletons.

You can see the experimental results for another problem called *party planning problem* in [31].

6. Related work

6.1. Parallel tree skeletons

Though trees are important data structures, writing general and efficient parallel programs manipulating trees is made hard by their irregular structures. This calls for helpful methods for parallel programming on trees and here the skeletal approach is a promising paradigm. As domain-specific skeletons, Deldari et al. [15] designed and implemented parallel skeletons for Constructive Solid Geometry (CSG). For

[2] You can download the source code of the binary-tree skeletons and the rose-tree skeletons from our project's website.

general-purpose tree skeletons, Skillicorn [11] first formalized a set of binary-tree skeletons based on Constructive Algorithmics [17–19]. The implementations of these binary-tree skeletons have been developed [12–14] based on the tree contraction algorithms [24,25].

For general trees or general recursive types, Skillicorn [11] and Ahn et al. [33] have given specifications of skeletons. However, the former lacks expressiveness whereas the latter did not give parallel implementations. We have tackled these problems and proposed a new set of rose-tree skeletons with efficient implementation. We believe our rose-tree skeletons are not only theoretically simple but also practically expressive.

6.2. Parallel computation on rose trees and nested lists

Parallel tree contraction algorithms are now the bases for efficient parallel computations on trees. Though the original idea proposed by Miller and Reif [25] did not limit the shapes of trees as binary trees, many researchers have developed more efficient tree contraction algorithms based on the assumption of binary trees [24,27,28,34]. Thus, according to the efficient tree contraction algorithms on binary trees, several studies developed parallel algorithms after representing the rose trees as binary trees. Cole and Vishkin [34], Diks and Hagerup [35], Skillicorn [11] and our previous paper [23] represented rose trees as binary trees by inserting dummy nodes to expand internal nodes. Though these representations suit specifying bottom–up and top–down computations, they are poor at specifying intra-siblings computations. In this paper, we adopt another binary-tree representation [29], and this enables us to formalize and implement intra-siblings computations as well. As we see in the example of the preorder numbering problem, these intra-siblings computations play important roles in the manipulation of rose trees.

Several researchers have studied hard the parallel manipulations of nested lists. NESL [36] provides computational patterns for nested computations, and Palmer et al. discussed how nested computations can be compiled on this paradigm [37]. We can consider nested lists as a subset of rose trees, and the idea of intra-siblings computations comes from these nested lists.

7. Conclusion

In this paper, we have proposed a set of parallel rose-tree skeletons. We designed these skeletons as simple as possible based on Constructive Algorithmics, and showed a parallel implementation based on the binary-tree representation. Our rose-tree skeletons are natural extensions of the binary-tree skeletons proposed so far, and we have added two computational patterns to denote computations among siblings. We have made a prototype implementation of our parallel rose-tree skeletons on our binary-tree skeleton library. Despite the rapid development, the skeletons have shown good scalability.

We hope that our rose-tree skeletons provide a basis for not only designing but also implementing parallel tree programs. Our future work is to extend our skeletons with more involved computational patterns on rose trees. One of them is a tree manipulation with the sorting of siblings discussed by Diks and Hagerup [35].

References

[1] M. Cole, Algorithmic skeletons: a structured approach to the management of parallel computation, Research Monographs in Parallel and Distributed Computing, Pitman, London, 1989.
[2] F. Rabhi, S. Gorlatch, Patterns and Skeletons for Parallel and Distributed Computing, Springer-Verlag Inc., New York, 2002.
[3] G.E. Blelloch, Scans as primitive operations, IEEE Transactions on Computers 38 (11) (1989) 1526–1538.
[4] W. Chin, A. Takano, Z. Hu, Parallelization via context preservation, IEEE Computer Society International Conference on Computer Languages (ICCL'98) (1998) 153–162.
[5] M. Cole, Parallel programming with list homomorphisms, Parallel Processing Letters 5 (2) (1995) 191–203.
[6] S. Gorlatch, Systematic efficient parallelization of scan and other list homomorphisms, in: EuroPar '96 Parallel Processing, Second International Euro-Par Conference, Lecture Notes in Computer Science 1124, Springer-Verlag, LIP, ENS Lyon, France, 1996, pp. 401–408.
[7] H. Iwasaki, Z. Hu, A new parallel skeleton for general accumulative computations, International Journal of Parallel Programming 32 (5) (2004) 389–414.
[8] D.B. Skillicorn, The Bird–Meertens formalism as a parallel model, in: J.S. Kowalik, L. Grandinetti (Eds.), NATO ASI Workshop on Software for Parallel Computation, NATO ARW "Software for Parallel Computation", 106 of F, Springer-Verlag NATO ASI, Cetraro, Italy, 1992.

[9] D.B. Skillicorn, Foundations of Parallel Programming, Cambridge University Press, 1994.
[10] K. Matsuzaki, Z. Hu, M. Takeichi, Parallelization with tree skeletons, in: Proceedings of the Ninth EuroPar Conference (EuroPar 2003), Lecture Notes in Computer Science 2790, Springer-Verlag, Klagenfurt, Austria, 2003, pp. 789–798.
[11] D.B. Skillicorn, Parallel implementation of tree skeletons, Journal of Parallel and Distributed Computing 39 (2) (1996) 115–125.
[12] J. Gibbons, Computing downwards accumulations on trees quickly, in: G. Gupta, G. Mohay, R. Topor (Eds.), Proceedings of 16th Australian Computer Science Conference, vol. 15 (1), Australian Computer Science Communications, 1993, pp. 685–691.
[13] J. Gibbons, W. Cai, D.B. Skillicorn, Efficient parallel algorithms for tree accumulations, Science of Computer Programming 23 (1) (1994) 1–18.
[14] K. Matsuzaki, Z. Hu, M. Takeichi, Implementation of parallel tree skeletons on distributed systems, in: Proceedings of the Third Asian Workshop on Programming Languages and Systems, Shanghai, China, 2002, pp. 258–271.
[15] H. Deldari, J.R. Davy, P.M. Dew, A skeleton for parallel CSG with a performance model, Tech. Rep., School of Computer Studies Research Report Series, University of Leeds, 1997.
[16] L. Meertens, First steps towards the theory of rose trees, CWI, Amsterdam, IFIP Working Group 2.1 Working paper 592 ROM-25, 1988.
[17] R.S. Bird, An introduction to the theory of lists, in: M. Broy (Ed.), Logic of Programming and Calculi of Discrete Design, NATO ASI Series F, 36, Springer-Verlag, 1987, pp. 5–42.
[18] R.S. Bird, O. de Moor, Algebras of Programming, Prentice Hall, 1996.
[19] J. Jeuring, Theories for algorithm calculation, Ph.D thesis, Faculty of Science, Utrecht University, parts of the thesis appeared in the Lecture Notes of the STOP 1992 Summer school on Constructive Algorithmics (1993).
[20] K. Emoto, Z. Hu, K. Kakehi, M. Takeichi, A compositional framework for developing parallel programs on two dimensional arrays, Tech. Rep. METR2005-09, Department of Mathematical Informatics, University of Tokyo, 2005.
[21] R. Bird, Introduction to Functional Programming using Haskell, Prentice Hall, 1998.
[22] S. Peyton Jones, J. Hughes, Report on the programming language Haskell 98: a non-strict, purely functional language. Available from: <http://www.haskell.org/> (1999).
[23] K. Matsuzaki, Z. Hu, K. Kakehi, M. Takeichi, Systematic derivation of tree contraction algorithms, Parallel Processing Letters 15 (3) (2005) 321–336.
[24] K. Abrahamson, N. Dadoun, D.G. Kirkpatrik, T. Przytycka, A simple parallel tree contraction algorithm, Journal of Algorithms 10 (2) (1989) 287–302.
[25] G.L. Miller, J.H. Reif, Parallel tree contraction and its application, in: 26th Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press, Portland, OR, 1985, pp. 478–489.
[26] K. Matsuzaki, Z. Hu, M. Takeichi, Design and implementation of general tree skeletons, Tech. Rep. METR2005-30, Department of Mathematical Engineering and Information Physics, University of Tokyo, 2005.
[27] D.A. Bader, S. Sreshta, N.R. Weisse-Bernstein, Evaluating arithmetic expressions using tree contraction: a fast and scalable parallel implementation for symmetric multiprocessors (SMPs), in: Ninth International Conference on High Performance Computing (HiPC 2002), Lecture Notes in Computer Science 2552, Bangalore, India, 2002, pp. 63–75.
[28] E.W. Mayr, R. Werchner, Optimal tree contraction and term matching on the hypercube and related networks, Algorithmica 18 (3) (1997) 445–460.
[29] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, second ed., MIT Press, 2001.
[30] SkeTo project homepage. Available from: <http://www.ipl.t.u-tokyo.ac.jp/sketo/>, 2005.
[31] K. Matsuzaki, K. Emoto, H. Iwasaki, Z. Hu, A library of constructive skeletons for sequential style of parallel programming, in: First International Conference on Scalable Information Systems (INFOSCALE 2006), Hong Kong, May 29–June 1, 2006.
[32] G. Kazai, M. Lalmas, N. Fuhr, N. Gövert, A report on the first year of the INitiative for the Evaluation of XML retrieval (INEX'02), Journal of the American Society for Information Science and Technology 55 (6) (2002).
[33] J. Ahn, T. Han, An analytical method for parallelization of recursive functions, Parallel Processing Letters 10 (1) (2000) 87–98.
[34] R. Cole, U. Vishkin, The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time, Algorithmica 3 (1988) 329–346.
[35] K. Diks, T. Hagerup, More general parallel tree contraction: register allocation and broadcasting in a tree, Theoretical Computer Science 203 (1) (1998) 3–29.
[36] G.E. Blelloch, J.C. Hardwick, J. Sipelstein, M. Zagha, S. Chatterjee, Implementation of a portable nested data-parallel language, Journal of Parallel and Distributed Computing 21 (1) (1994) 4–14.
[37] D. Palmer, J. Prins, S. Chatterjee, R. Faith, Piecewise execution of nested data-parallel programs, in: Languages and Compilers for Parallel Computing: Eighth International Workshop, Columbus, OH, USA, August 10–12, 1995 Proceedings, Lecture Notes in Computer Science 1033, Springer-Verlag, 1996, pp. 346–361.