

Systematic Derivation of Tree Contraction Algorithms

Kiminori Matsuzaki¹, Zhenjiang Hu^{1,2}, Kazuhiko Kakehi¹, and Masato Takeichi¹

¹ Graduate School of Information Science and Technology,
University of Tokyo

{Kiminori.Matsuzaki,hu,kaz,takeichi}@mist.i.u-tokyo.ac.jp

² PRESTO21, Japan Science and Technology Cooperation

Abstract. While tree contraction algorithms play an important role in efficient tree computation in parallel, it is difficult to develop such algorithms due to the strict conditions imposed on contracting operators. In this paper, we propose a systematic method of deriving efficient tree contraction algorithms from recursive functions on trees in any shape. We identify a general recursive form that can be parallelized to obtain efficient tree contraction algorithms, and present a derivation strategy for transforming general recursive functions to parallelizable form. We illustrate our approach by deriving a novel parallel algorithm for the maximum connected-set sum problem on arbitrary trees, the tree-version of the famous maximum segment sum problem.

Keywords. Tree Contraction, Parallelization, Skeletal Parallelism, Rose Tree, Maximum Segment Sum Problem.

1 Introduction

Skeletal parallel programming [5, 17] is an elegant model for developing efficient and correct parallel programs. Although many researchers have devoted themselves to the algorithmic skeletons on lists [6, 9, 12, 19], not very many studies have been addressed to other datatypes such as trees and graphs.

Trees are important datatypes, widely used in representing structured documents such as XML. There are two approaches to parallel computation on trees, the first is the *divide and conquer* approach [2], and the second is the *tree contraction* approach [1, 15, 16, 18]. The divide and conquer approach simply computes each child tree independently, and its parallel cost is $O(h + w)$, where h denotes the height of a tree and w denotes the nodes' maximum number of children. Therefore, it may be very inefficient if the tree is ill-balanced or a node has too many children. By contrast, the tree contraction approach provides efficient parallel algorithms even for ill-balanced trees. The well-known algorithm, the *shunt contraction*, can run on binary trees in logarithmic time to their size. However, it requires the tree contracting operators that have to meet the closure property to be intelligently designed, which is known to be hard, and thus discourages programmers from using it.

Some attempts have been made on formal specifications for parallel tree algorithms. Gibbons et al. [8] and Skillicorn [20, 21] defined four skeletons on binary trees and gave an efficient implementation of them based on the tree contraction algorithm. Skillicorn also showed the usefulness of these skeletons with some examples of the manipulation of structured documents [20, 22, 23]. Deldari et al. [7] designed a skeleton for constructive solid geometry. Matsuzaki et al. [14] proposed a systematic method of composing efficient parallel programs in terms of the skeletons on binary trees. However, there have really been very few studies on the formal derivation of parallel tree algorithms.

In this paper, we consider the parallelization of a general tree recursive function, called (tree) reduction, which can concisely specify the computation of calculating a value through a bottom-up traversal of the tree. Informally, function f is a reduction, if it is defined in the

following recursive form:

$$\begin{aligned} f (RLeaf\ a) &= k_1\ a \\ f (RNode\ b\ [t_1, t_2, \dots, t_n]) &= k_2\ b\ [f\ t_1, f\ t_2, \dots, f\ t_n], \end{aligned}$$

where k_1 and k_2 are two functions. As discussed in Skillicorn [21, 23], certain conditions on k_2 are necessary for the existence of efficient parallel algorithms. One condition proposed so far in [13, 21, 23] is to define k_2 in terms of associative operator \oplus as follows.

$$\begin{aligned} reduce\ (\oplus)\ (RLeaf\ a) &= k'_1\ a \\ reduce\ (\oplus)\ (RNode\ b\ [t_1, t_2, \dots, t_n]) \\ &= k'_2\ b\ \oplus\ reduce\ (\oplus)\ t_1\ \oplus\ reduce\ (\oplus)\ t_2\ \oplus\ \dots\ \oplus\ reduce\ (\oplus)\ t_n \end{aligned}$$

This definition is easy to understand but lacks expressiveness. To demonstrate this, consider developing an efficient parallel program for XML serialization, which accepts an XML tree and returns its tagged-formatted string. We may solve this problem with the following recursive definition:

$$\begin{aligned} x2s\ (RLeaf\ a) &= a \\ x2s\ (RNode\ b\ [t_1, t_2, \dots, t_n]) &= tags\ b\ \oplus\ (x2s\ t_1 \# x2s\ t_2 \# \dots \# x2s\ t_n) \\ &\quad \text{where } tags\ b = (\text{“<”} \# b \# \text{“>”}, \text{“</”} \# b \# \text{“>”}) \\ &\quad (s, e) \oplus t = s \# t \# e, \end{aligned}$$

where $\#$ is an infix-operator to concatenate two strings. It is not obvious, however, how to define $x2s$ in terms of $reduce$, because we need two different binary operators, namely $\#$ and \oplus , to define k_2 .

In this paper, we aim at a systematic method of parallelizing a class of useful reductions to ones that can be efficiently implemented by tree contraction. Our method can deal with recursive definitions in which k_2 is defined using two binary operators. The contributions this paper makes can be summarized as follows:

- We give a new formalization of the condition for shunt contraction (Theorem 1), which is more constructive in the sense that tree contracting operators can be automatically derived from semantic conditions. In addition, to eliminate the limitation where the shunt contraction can only be applied to binary trees, we show how to transform rose trees (trees whose nodes can have an arbitrary number of children) to binary trees so that shunt contraction can be applied.
- We not only recognize the importance of distributivity in the derivation of tree contraction algorithms, but also give an extension of distributivity that is suited to systematic derivation with generalization and context-preserving transformation. We particularly identify a general recursive form that can be parallelized (Theorem 2), and highlight a derivation strategy for transforming general recursive functions to parallelizable form.
- We demonstrate the effectiveness of our approach by deriving an efficient parallel program for the tree version of the maximum segment sum problem [3]. Much work has been done on the parallelization of the problem: on lists [6, 10], on 2-dimensional arrays [11], and on binary trees [14]. To the best of our knowledge, this is the first derivation of the parallel program for rose trees, the most complex data structure ever.

This paper is organized as follows. After reviewing the notational conventions and datatypes, we show how an arbitrary tree is arranged in the form of a binary tree in Section 2. We then formalize the conditions for tree contraction in a more constructive way in Section 3, and give a property which is an extension of distributivity in Section 4. In Section 5, we give a definition of parallelizable reductions on rose trees, and show how these reductions are parallelized. Then, we propose a strategy for systematic parallelization and demonstrate the effectiveness of our approach with a non-trivial example, namely the maximum connected-sum problem, in Section 6. Finally, we make some concluding remarks.

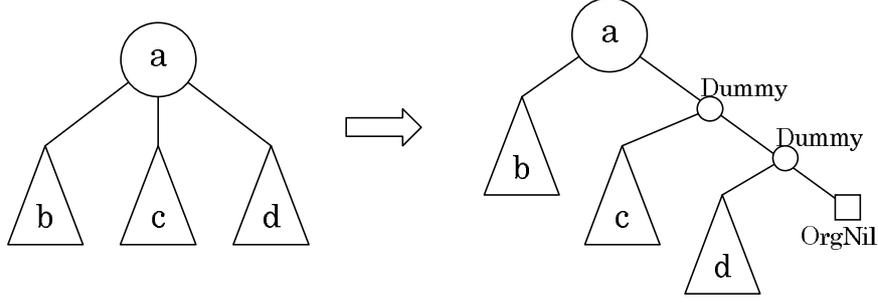


Fig. 1. Local rearrangement from a rose tree into a binary tree

2 Preliminaries

2.1 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 not $f (a \oplus b)$. Infix binary operators will be denoted by \oplus , \otimes , and their units are written as ι_{\oplus} , ι_{\otimes} , respectively, in this paper.

2.2 Datatypes

The *cons list* is constructed with an empty list or by adding an element to a list. The datatype for a list where every element has type α is defined as follows.

$$\text{data List } \alpha = \text{Nil} \mid \text{Cons } \alpha (\text{List } \alpha)$$

We may use abbreviations, i.e., $[\alpha]$ for datatype $\text{List } \alpha$, $[]$ for Nil , and $(a : as)$ for $\text{Cons } a as$.

A binary tree is a tree whose internal nodes have exactly two children. The datatype for binary trees where every leaf has type α and every internal node has type β is defined as follows.

$$\text{data BTree } \alpha \beta = \text{Leaf } \alpha \mid \text{Node } \beta (\text{BTree } \alpha \beta) (\text{BTree } \alpha \beta)$$

A rose tree is a tree whose internal nodes have an arbitrary number of children. The datatype for rose trees where every leaf has type α and every internal node has type β is defined using a list as follows.

$$\text{data RTree } \alpha \beta = \text{RLeaf } \alpha \mid \text{RNode } \beta [\text{RTree } \alpha \beta]$$

2.3 Representation of Rose Trees

Since the tree contraction algorithm only accepts binary trees, rose trees ought to be held in the shape of binary trees. In this paper, we will use the arrangement (representation) in Fig. 1. This arrangement turns the leaf and internal node of a rose tree into a leaf and the root node of the corresponding subtree in the binary tree, respectively. Some dummy nodes are inserted into this binary tree to unroll the children and to represent the children's end. This is almost the same arrangement as in [21], and there have been some discussions about the implementation of the tree contraction algorithm on these arranged binary trees.

To formally define the arrangement, we initially define two new types.

$$\begin{aligned} \text{data R2BLeaf } \alpha &= \text{OrgLeaf } \alpha \mid \text{OrgNil} \\ \text{data R2BNode } \beta &= \text{OrgNode } \beta \mid \text{Dummy} \end{aligned}$$

$R2BLeaf$ represents the types of leaves in the binary tree, and is constructed with the leaf in the rose tree ($OrgLeaf$) or the sentinel for the end of the children ($OrgNil$). $R2BNode$ represents the types of internal nodes in the binary tree, and is constructed by the internal node in the rose tree ($OrgNode$) or the dummy node inserted to expand the children ($Dummy$). The function $r2b$, which performs this arrangement, can be formally defined using auxiliary function $r2b'$, as follows.

$$\begin{aligned}
r2b &:: RTree \alpha \beta \rightarrow BTree (R2BLeaf \alpha) (R2BNode \beta) \\
r2b (RLeaf a) &= Leaf (OrgLeaf a) \\
r2b (RNode b (x : xs)) &= Node (OrgNode b) (r2b x) (r2b' xs) \\
r2b' [] &= Leaf OrgNil \\
r2b' (x : xs) &= Node Dummy (r2b x) (r2b' xs)
\end{aligned}$$

Below, we briefly analyze the number of additional nodes in the binary trees after the above transformation. Let n_l be the number of leaves, and n_{in} be the number of internal nodes in an input rose tree. The binary tree transformed from the rose tree has $2n_l + 2n_{in} - 1$ nodes. In brief, the transformed binary tree has $2n - 1$ nodes, where n is the number of nodes in the original rose tree. Consequently, by using tree contraction algorithms, we can also compute on rose trees in logarithmic parallel time.

3 Tree Contraction Algorithm and its Derivation

Tree contraction algorithms are efficient parallel algorithms to reduce trees. Of the tree contraction algorithms, shunt contraction [1] is widely known as a simple and efficient algorithm on EREW PRAM. The shunt contraction algorithm accepts binary trees, and reduces them with two symmetric operations, namely $ContractL$ and $ContractR$. $ContractL/ContractR$ operation replaces an internal node, its left/right leaf, and its right/left node, with a new node.

In the following, we assume reduction on the binary tree is defined as follows.

$$\begin{aligned}
f (Leaf a) &= k_1 a \\
f (Node b t_1 t_2) &= k_2 b (f t_1) (f t_2)
\end{aligned}$$

To guarantee the overall logarithmic parallel cost of the shunt contraction algorithm, each local contraction must be done in constant time and space. Such conditions are given in [1] as follows.

- For every internal node $Node b t_1 t_2$, sectioned function $k_2 b$ is drawn from an indexed set of functions G that contains the identity function.
- All functions in G can be applied in constant time.
- If g_i and g_j are functions in G , for any value l or r , then the two functions $\lambda xy.g_i l (g_j x y)$ and $\lambda xy.g_i (g_j x y) r$ are in G , and their indices can be computed from indices i and j and values l or r in constant time.

These conditions are too abstract for the programmer to check or use for derivation.

To enable systematic derivation, we introduced the idea of parametrized functions. We used the notation $G[a]$ for a function embodied from a set of parametrized functions G with parameter a . For example, if the set of parametrized functions is given as $G = \{ \lambda xy. a + x + y \}$, then functions $G[1] = \lambda xy. 1 + x + y$ and $G[2] = \lambda xy. 2 + x + y$ are the embodiments with 1 and 2, respectively.

Let us now restrict the indexed set of functions to the set of parametrized functions. Although some algorithms, in which different functions are applied to internal nodes, may be

1. Number the leaves left to right beginning at 0.
2. Initialize every leaf and internal node by applying ψ_1 and ψ_2 , respectively.
3. Iterate until a node remains.
 - (a) For every left leaf whose index is even, perform ContractL. If the other child is a leaf apply a function embodied from G with parent's value n , or otherwise apply ϕ_L .
 - (b) For every right leaf whose index is even and not involved in the previous step, perform ContractR. If the other child is a leaf apply $G[n]$, or otherwise apply ϕ_R .
 - (c) Renumber the leaves by dividing their indices by two and rounding down.

Fig. 2. Shunt Contraction Algorithm Based on Parametrized Functions

unacceptable under this restriction, numerous tree algorithms represented as tree reductions can be dealt with. With the notation of parametrized functions, sufficient conditions for shunt contraction are given by the following theorem.

Theorem 1. If there are a set of parametrized functions G with the identity function, and three functions ψ_2 , ϕ_L and ϕ_R such that the following conditions are satisfied, then we can use the shunt contraction algorithm.

- For every internal node *Node* b t_1 t_2 , sectioned function k_2 b is semantically equivalent to function $G[\psi_2$ $b]$.
- All the functions embodied from G can be applied in constant time.
- For any parameters a_1 and a_2 , and any values l and r , the following equations

$$\begin{aligned} \lambda xy.G[a_1] l (G[a_2] x y) &= \lambda xy.G[\phi_L a_1 l a_2] x y \\ \lambda xy.G[a_1] (G[a_2] x y) r &= \lambda xy.G[\phi_R a_1 r a_2] x y \end{aligned}$$

hold, and ϕ_L and ϕ_R are computed in constant time. □

If a tree algorithm meets these conditions, we can utilize the shunt contraction algorithm using the functions above, G , ψ_2 , ϕ_L , ϕ_R , and function ψ_1 ($= k_1$) as shown in Fig. 2. Therefore, we only have to derive the set of parametrized functions G and functions ψ_1 , ψ_2 , ϕ_L and ϕ_R . To demonstrate how Theorem 1 works, let us illustrate it with a very simple program.

Example 1. A recursive program that computes the sum of values for all nodes is given as follows.

$$\begin{aligned} \text{sumtree} (\text{Leaf } a) &= a \\ \text{sumtree} (\text{Node } b \ t_1 \ t_2) &= b + \text{sumtree } t_1 + \text{sumtree } t_2 \end{aligned}$$

An adequate definition of the set of parametrized functions G is given with parameter a as $G = \{ \lambda xy.a + x + y \}$. From the definition above, the initializing functions are $\psi_1 = id$ and $\psi_2 = id$, where id is the identity function. The contracting operations ϕ_L and ϕ_R become $\phi_L a_1 l a_2 = a_1 + l + a_2$ and $\phi_R a_1 r a_2 = a_1 + r + a_2$. With Theorem 1, we can utilize the tree contraction algorithm as shown in Fig. 2 using these functions. □

4 Extension of Distributive Law

Before discussing the parallelization of reductions, let us now discuss generalization of the distributive law. It is well known that associativity and distributivity play important roles in parallelizing programs. For example, the distributivity of \times over $+$ enables us to simplify the expression as:

$$1 + 2 \times (3 + 4 \times x) = 1 + 2 \times 3 + 2 \times 4 \times x = 7 + 8 \times x .$$

Borrowing the idea of *contexts* or normal forms from [4], we extend the characteristic of normalization to derive parallel program over two operators.

Definition 1. Let operator \otimes be associative. The function defined with two operators, \otimes and \oplus , is said to be in *distributive normal form*, if it is written as

$$\lambda x. a \oplus (b \otimes x \otimes c) ,$$

where a , b , and c are constants. □

Definition 2. Operator \otimes is said to be *extended-distributive* over \oplus , if the distributive normal form is preserved over function composition. In other words, there are appropriate functions p_1 , p_2 , and p_3 , and for any a_1 , b_1 , c_1 , a_2 , b_2 , and c_2 , the following equation holds:

$$(\lambda x. a_1 \oplus (b_1 \otimes x \otimes c_1)) \circ (\lambda x. a_2 \oplus (b_2 \otimes x \otimes c_2)) = \lambda x. A \oplus (B \otimes x \otimes C) ,$$

where A , B , and C are computed with $A=p_1(a_1, b_1, c_1, a_2, b_2, c_2)$, $B=p_2(a_1, b_1, c_1, a_2, b_2, c_2)$, and $C = p_3(a_1, b_1, c_1, a_2, b_2, c_2)$. These functions p_1 , p_2 , and p_3 are called *characteristic functions*. □

Although the definition of extended-distributivity is complex, it has many applications. We can uniformly use this property for the associative operator, the distributive operator, or other operators as demonstrated in the following examples. In Example 4, we demonstrate how to derive characteristic functions from the definition.

Example 2. Extended-distributivity can replace associativity. Let operator \oplus be the same as associative operator \otimes . Then, \otimes is extended-distributive over \oplus ($= \otimes$) and the characteristic functions are as follows.

$$\begin{aligned} p_1(a_1, b_1, c_1, a_2, b_2, c_2) &= a_1 \otimes b_1 \otimes a_2 \otimes b_2 \\ p_2(a_1, b_1, c_1, a_2, b_2, c_2) &= \iota_{\otimes} \\ p_3(a_1, b_1, c_1, a_2, b_2, c_2) &= c_2 \otimes c_1 \end{aligned} \quad \square$$

Example 3. Extended-distributivity is a generalization of the distributive law. Let two operators \otimes and \oplus constitute the ring, that is, let \oplus be associative and \otimes be not only associative but also distributive over \oplus . Then \otimes is extended-distributive over \oplus and the characteristic functions are as follows.

$$\begin{aligned} p_1(a_1, b_1, c_1, a_2, b_2, c_2) &= a_1 \oplus (b_1 \otimes a_2 \otimes c_1) \\ p_2(a_1, b_1, c_1, a_2, b_2, c_2) &= b_1 \otimes b_2 \\ p_3(a_1, b_1, c_1, a_2, b_2, c_2) &= c_2 \otimes c_1 \end{aligned} \quad \square$$

To evaluate the extended-distributivity and derive the characteristic functions, we calculate and verify that two expressions E_1 and E_2 defined as

$$\begin{aligned}\lambda x. E_1 &= (\lambda x. a_1 \oplus (b_1 \otimes x \otimes c_1)) \circ (\lambda x. a_2 \oplus (b_2 \otimes x \otimes c_2)) \\ &= \lambda x. a_1 \oplus (b_1 \otimes (a_2 \oplus (b_2 \otimes x \otimes c_2)) \otimes c_2) \\ \lambda x. E_2 &= \lambda x. A \oplus (B \otimes x \otimes C)\end{aligned}$$

have the same form by substituting proper expressions for the capital parameters. To demonstrate the derivation of characteristic functions, we show that operator \oplus is extended-distributive over \oplus in the definition of $x2s$ in the introduction, and derive the characteristic functions.

Example 4. Let operator \oplus be defined with associative operator $\#$ as $(s, e) \oplus t = s \# t \# e$. This operator \oplus is not distributive over \oplus as is easily demonstrated as follows:

$$\begin{aligned}(s, e) \oplus (x \# y) &= s \# x \# y \# e \\ ((s, e) \oplus x) \# ((s, e) \oplus y) &= s \# x \# e \# s \# y \# e\end{aligned}$$

To verify extended-distributivity, we first expand the two expressions E_1 and E_2 above.

$$\begin{aligned}E_1 &= (s_1, e_1) \oplus (t_1 \# ((s_2, e_2) \oplus (t_2 \# x \# t'_2)) \# t'_1) \\ &= (s_1, e_1) \oplus (t_1 \# s_2 \# t_2 \# x \# t'_2 \# e_2 \# t'_1) \\ &= s_1 \# t_1 \# s_2 \# t_2 \# x \# t'_2 \# e_2 \# t'_1 \# e_1 \\ E_2 &= (S, E) \oplus (T \# x \# T') \\ &= S \# T \# x \# T' \# E\end{aligned}$$

From calculation result above, the correspondences of capital variables are,

$$\begin{aligned}S \# T &= s_1 \# t_1 \# s_2 \# t_2, \text{ and} \\ T' \# E &= t'_2 \# e_2 \# t'_1 \# e_1.\end{aligned}$$

There are many solutions to the equations above, and one of those is as follows, which can also be considered as a set of characteristic functions.

$$\begin{aligned}p_1((s_1, e_1), t_1, t'_1, (s_2, e_2), t_2, t'_2) &= (S, E) = (s_1 \# t_1 \# s_2 \# t_2, t'_2 \# e_2 \# t'_1 \# e_1) \\ p_2((s_1, e_1), t_1, t'_1, (s_2, e_2), t_2, t'_2) &= T = [] \\ p_3((s_1, e_1), t_1, t'_1, (s_2, e_2), t_2, t'_2) &= T' = []\end{aligned}$$

We can also show extended-distributivity and derive the characteristic functions for general \oplus defined with associative operator \otimes . \square

If operator \otimes is also commutative, then we can simplify the definitions for the distributive normal form and extended-distributivity. Here, distributive normal form $\lambda xy. a \oplus (b \otimes x \otimes c)$ can be simplified to $\lambda xy. a \oplus (b' \otimes x)$ by reversing x and c , and substituting b' for $b \otimes c$. Extended-distributivity is also defined in this form, and we say \otimes is extended-distributive over \oplus if there are appropriate functions p_1 and p_2 such that for any $a_1, b_1, a_2,$ and b_2 the following equation holds. The characteristic functions are minimized into two functions p_1 and p_2 in this case.

$$\begin{aligned}(\lambda x. a_1 \oplus (b_1 \otimes x)) \circ (\lambda x. a_2 \oplus (b_2 \otimes x)) &= \lambda x. A \oplus (B \otimes x) \\ \text{where } A &= p_1(a_1, b_1, a_2, b_2) \\ B &= p_2(a_1, b_1, a_2, b_2)\end{aligned}$$

5 Parallelizable Reduction

In this section, we present a class of reductions that can be systematically parallelized based on the tree contraction algorithm. A reduction represents a class of computation which collapses the tree into a single value, and the general definition for reduction is as follows.

$$\begin{aligned} f (RLeaf a) &= k_1 a \\ f (RNode b [t_1, t_2, \dots, t_n]) &= k_2 b [f t_1, f t_2, \dots, f t_n] \end{aligned}$$

Definition 3. Let \otimes be an associative operator. A function is said to be *parallelizable reduction*, if the function is defined in the following form.

$$\begin{aligned} f (RLeaf a) &= k_1 a \\ f (RNode b [t_1, t_2, \dots, t_n]) &= k_2 b \oplus (f t_1 \otimes f t_2 \otimes \dots \otimes f t_n) \end{aligned}$$

We can rephrase this using auxiliary function f' more formally.

$$\begin{aligned} f (RLeaf a) &= k_1 a \\ f (RNode b ts) &= k_2 b \oplus f' ts \\ f' [] &= \iota_{\otimes} \\ f' (t : ts) &= f t \otimes f' ts \end{aligned} \quad \square$$

Parallelizable reduction is defined in two steps for each node. First, the siblings are collapsed with associative operator \otimes , which is the same operation as the reduction on lists. Then, computation on the previous result and parent are done with another operator \oplus . We can write many reductions in this form, for example, the XML serialization that was in the Introduction, the sum of values for all nodes, and the height of the tree.

In the following, we will demonstrate that parallelizable reduction can efficiently be computed with the tree contraction algorithm on arranged binary trees. Let the set of parametrized functions G be defined as:

$$G[(a, b, c)] = \lambda xy. a \oplus (b \otimes x \otimes y \otimes c) .$$

Using the embodiments of this set of parametrized functions G , we can describe new function h on the arranged binary trees as follows.

$$\begin{aligned} h (Leaf (OrgLeaf a)) &= k_1 a \\ h (Leaf OrgNil) &= \iota_{\otimes} \\ h (Node (OrgNode b) t_1 t_2) &= G[(k_2 b, \iota_{\otimes}, \iota_{\otimes})] (h t_1) (h t_2) \\ h (Node Dummy t_1 t_2) &= G[(\iota_{\oplus}, \iota_{\otimes}, \iota_{\otimes})] (h t_1) (h t_2) \end{aligned}$$

Let us first confirm the correctness of the computation of h on the arranged binary trees.

Lemma 1. Function h defined above satisfies $h \circ r2b = f$, $h \circ r2b' = f'$.

Proof: We can prove this lemma by structural induction over the rose tree: base cases for $RLeaf a$ and $[]$, and inductive cases for $RNode b (t : ts)$ and $(t : ts)$, respectively. \square

Next, let us confirm that the set of parametrized functions G satisfies the conditions for the tree contraction algorithm.

Lemma 2. Let \otimes be an associative operator and be distributive over \oplus with the characteristic functions p_1 , p_2 , and p_3 . Then, for any parameters $a_1, b_1, c_1, a_2, b_2, c_2$, and values l and r ,

$$\begin{aligned} \lambda xy. G[(a_1, b_1, c_1)] l (G[(a_2, b_2, c_2)] x y) &= \lambda xy. G[\phi_L (a_1, b_1, c_1) l (a_2, b_2, c_2)] x y \\ \lambda xy. G[(a_1, b_1, c_1)] (G[(a_2, b_2, c_2)] x y) r &= \lambda xy. G[\phi_R (a_1, b_1, c_1) r (a_2, b_2, c_2)] x y \end{aligned}$$

holds for appropriate functions ϕ_L and ϕ_R .

Proof: We can define the two functions ϕ_L and ϕ_R using \otimes , p_1 , p_2 , and p_3 as follows.

$$\begin{aligned} \phi_L (a_1, b_1, c_1) l (a_2, b_2, c_2) &= (p_1 \text{ tup}_L, p_2 \text{ tup}_L, p_3 \text{ tup}_L) \\ &\quad \mathbf{where} \text{ tup}_L = (a_1, b_1 \otimes l, c_1, a_2, b_2, c_2) , \\ \phi_R (a_1, b_1, c_1) r (a_2, b_2, c_2) &= (p_1 \text{ tup}_R, p_2 \text{ tup}_R, p_3 \text{ tup}_R) \\ &\quad \mathbf{where} \text{ tup}_R = (a_1, b_1, r \otimes c_1, a_2, b_2, c_2) . \end{aligned}$$

Then, we can prove these two equations with simple calculations. \square

Theorem 2. Function f defined in Definition 3 can be parallelized by the tree contraction algorithm on binary trees as arranged in Section 2.3, if operator \otimes is associative and extended-distributive over \oplus .

Proof: Since operator \otimes is associative and extended-distributive over \oplus , we assume that the characteristic functions of extended-distributivity are p_1 , p_2 , and p_3 . We can construct the initialize functions ψ_1 and ψ_2 , the contracting operations ϕ_L and ϕ_R , and the set of functions G in the following way. In the rest of this paper, due to space limitations, we will place the definitions of ψ_1 and ψ_2 side by side.

$$\begin{aligned} \psi_1 (\text{OrgLeaf } a) &= k_1 a & \psi_2 (\text{OrgNode } b) &= (k_2 b, \iota_{\otimes}, \iota_{\otimes}) \\ \psi_1 \text{OrgNil} &= \iota_{\otimes} & \psi_2 \text{Dummy} &= (\iota_{\oplus}, \iota_{\otimes}, \iota_{\otimes}) \\ \phi_L (a_1, b_1, c_1) l (a_2, b_2, c_2) &= (p_1 \text{ tup}_L, p_2 \text{ tup}_L, p_3 \text{ tup}_L) \\ \phi_R (a_1, b_1, c_1) r (a_2, b_2, c_2) &= (p_1 \text{ tup}_R, p_2 \text{ tup}_R, p_3 \text{ tup}_R) \\ &\quad \mathbf{where} \text{ tup}_L = (a_1, b_1 \otimes l, c_1, a_2, b_2, c_2) \\ &\quad \text{tup}_R = (a_1, b_1, r \otimes c_1, a_2, b_2, c_2) \\ G[(a, b, c)] &= \lambda xy. a \oplus (b \otimes x \otimes y \otimes c) \end{aligned}$$

It follows from Lemmas 1 and 2 that the theorem holds. \square

To illustrate an application of this theorem, let us derive a parallel algorithm from the definition of $x2s$ in the introduction.

Example 5. Function $x2s$ can be computed in parallel because operator $\#$ is not only associative but also extended-distributive over \oplus as mentioned in Example 4. We can derive a parallel program by utilizing the result of Example 4 for Theorem 2, and the derived program is as follows.

$$\begin{aligned} \psi_1 (\text{OrgLeaf } a) &= a & \psi_2 (\text{OrgNode } b) &= (\text{tags } b, [], []) \\ \psi_1 \text{OrgNil} &= [] & \psi_2 \text{Dummy} &= (([], []), [], []) \\ \phi_L ((s_1, e_1), t_1, t'_1) l ((s_2, e_2), t_2, t'_2) &= ((s_1 \# t_1 \# l \# s_2 \# t_2, t'_2 \# e_2 \# t'_1 \# e_1), [], []) \\ \phi_R ((s_1, e_1), t_1, t'_1) r ((s_2, e_2), t_2, t'_2) &= ((s_1 \# t_1 \# s_2 \# t_2, t'_2 \# e_2 \# r \# t'_1 \# e_1), [], []) \\ G[((s, e), t, t')] &= \lambda xy. s \# t \# x \# y \# t' \# e \end{aligned} \quad \square$$

In some cases, we can optimize the derived parallel program. If all the values for a variable in the definitions of ψ_2 , ϕ_L , and ϕ_R are the same, we can remove the variable after substituting the value into the definitions. For example, in the parallel program above, the second and the third variables, i.e. t and t' , are always the empty string, $[]$. Therefore we can remove the variables after substituting $[]$ for t_1 , t'_1 , t_2 , and t'_2 . The optimized program is as follows.

$$\begin{aligned} \psi_1 (\text{OrgLeaf } a) &= a & \psi_2 (\text{OrgNode } b) &= \text{tags } b \\ \psi_1 \text{OrgNil} &= [] & \psi_2 \text{Dummy} &= ([], []) \\ \phi_L (s_1, e_1) l (s_2, e_2) &= (s_1 \# l \# s_2, e_2 \# e_1) \\ \phi_R (s_1, e_1) r (s_2, e_2) &= (s_1 \# s_2, e_2 \# r \# e_1) \\ G[(s, e)] &= \lambda xy. s \# x \# y \# e \end{aligned}$$

We will give some specializations of Theorem 2 for the operators examined in Examples 2 and 3 in the following.

Corollary 1. Let operator \oplus in the parallelizable reduction be the same as \otimes . We can then utilize the tree contraction algorithm with the following functions.

$$\begin{aligned}
\psi_1 (\text{OrgLeaf } a) &= k_1 a & \psi_2 (\text{OrgNode } b) &= (k_2 b, \iota_{\otimes}) \\
\psi_1 \text{ OrgNil} &= \iota_{\otimes} & \psi_2 \text{ Dummy} &= (\iota_{\otimes}, \iota_{\otimes}) \\
\phi_L (a_1, b_1) l (a_2, b_2) &= (a_1 \otimes l \otimes a_2, b_2 \otimes b_1) \\
\phi_R (a_1, b_1) r (a_2, b_2) &= (a_1 \otimes a_2, b_2 \otimes r \otimes b_1) \\
G[(a, b)] &= \lambda xy. a \otimes x \otimes y \otimes b
\end{aligned}
\quad \square$$

Corollary 2. Let operators \otimes and \oplus in the parallelizable reduction constitute a ring, that is, let \otimes and \oplus be associative, and \otimes be distributive over \oplus . We can then utilize the tree contraction algorithm with the following functions.

$$\begin{aligned}
\psi_1 (\text{OrgLeaf } a) &= k_1 a & \psi_2 (\text{OrgNode } b) &= (k_2 b, \iota_{\otimes}, \iota_{\otimes}) \\
\psi_1 \text{ OrgNil} &= \iota_{\otimes} & \psi_2 \text{ Dummy} &= (\iota_{\oplus}, \iota_{\otimes}, \iota_{\otimes}) \\
\phi_L (a_1, b_1, c_1) l (a_2, b_2, c_2) &= (a_1 \oplus (b_1 \otimes l \otimes a_2 \otimes c_1), b_1 \otimes l \otimes b_2, c_2 \otimes c_1) \\
\phi_R (a_1, b_1, c_1) r (a_2, b_2, c_2) &= (a_1 \oplus (b_1 \otimes a_2 \otimes r \otimes c_1), b_1 \otimes b_2, c_2 \otimes r \otimes c_1) \\
G[(a, b, c)] &= \lambda xy. a \oplus (b \otimes x \otimes y \otimes c)
\end{aligned}
\quad \square$$

If operator \otimes is not only associative but also commutative, then we can derive a parallel program more simply as the following corollary shows.

Corollary 3. Let operator \otimes in the parallelizable reduction be both associative and commutative. If operator \otimes is extended-distributive over \oplus with characteristic functions p_1 and p_2 , then we can utilize the tree contraction algorithm with the following functions. Here, contracting operations ϕ_L and ϕ_R have the same definition, namely ϕ .

$$\begin{aligned}
\psi_1 (\text{OrgLeaf } a) &= k_1 a & \psi_2 (\text{OrgNode } b) &= k_2 b \\
\psi_1 \text{ OrgNil} &= \iota_{\otimes} & \psi_2 \text{ Dummy} &= (\iota_{\oplus}, \iota_{\otimes}) \\
\phi (a_1, b_1) x (a_2, b_2) &= (p_1 (a_1, b_1 \otimes x, a_2, b_2), p_2 (a_1, b_1 \otimes x, a_2, b_2)) \\
G[(a, b)] &= \lambda xy. a \oplus (b \otimes x \otimes y)
\end{aligned}
\quad \square$$

6 Parallelization Strategy

Although we have extended-distributivity and parallelizable reduction in hand, users' programs may not be exactly compatible with them. Even so, we can still derive parallel programs systematically with the following strategy.

1. *Write specification:* In the first step, we write the specification as a recursive function in the form of parallelizable reduction. In this step, the operators used in the function do not need to be associative or extended-distributive. We derive the program in the form of parallelizable reduction by applying calculational techniques such as tupling or normalization of conditions.
2. *Derive associative operator:* In the second step, we derive an associative operator for \otimes , by applying the parallelization techniques that have been proposed for lists, for example, the fusion and tupling technique proposed by Hu et al. [10] or the context preservation technique proposed by Chin et al. [4].

3. *Derive extended-distributive operator:* In the third step, we derive operator \oplus such that operator \otimes is extended-distributive over \oplus . We derive such an operator by iterated generalization and verification. To avoid inconsistency over the \otimes , we only generalize the definition of \oplus for the left argument in this step.
4. *Derive parallel program:* In the final step, we derive the contracting operations from the result for the previous step based on Theorem 2, and do some optimizations if possible.

In the following, to demonstrate the capability of our parallelization strategy, we demonstrate the derivation of an efficient parallel program for the maximum connected-set sum problem on trees with arbitrary shapes, which is the tree version of the maximum segment sum problem [3]. The maximum connected-set sum problem involves finding the maximum sum of all connected sets. A connected-set of a tree is a set of nodes where every two nodes are connected by following the nodes in the set.

Write the specification

We first write a recursive function for the problem. For the maximum connected-set sum problem, we can write a program using the dynamic programming technique, where the following two values are computed for each subtree.

- r : The maximum sum of all connected sets that include the root of the subtree. We can compute this value by adding the value of the root node to the sum of all positive r values of the root's immediate subtrees.
- s : The maximum sum of all connected sets that do not include the root of the subtree. We can compute this value by selecting the maximum r and s values of the root's immediate subtrees.

With this idea, we can define the following function, where operator \uparrow returns the larger value.

$$\begin{aligned}
mcs\ t &= \mathbf{let} \begin{pmatrix} r \\ s \end{pmatrix} = mcs'\ t \ \mathbf{in} \ r \uparrow s \\
mcs'\ (RLeaf\ a) &= \begin{pmatrix} a \\ 0 \end{pmatrix} \\
mcs'\ (RNode\ b\ [t_1, t_2, \dots, t_n]) &= b \oplus (g\ (mcs'\ t_1) \otimes g\ (mcs'\ t_2) \otimes \dots \otimes g\ (mcs'\ t_n)) \\
&\quad \mathbf{where} \ b \oplus \begin{pmatrix} r \\ s \end{pmatrix} = \begin{pmatrix} b+r \\ s \end{pmatrix} \\
&\quad g \begin{pmatrix} r \\ s \end{pmatrix} = \begin{pmatrix} r \uparrow 0 \\ r \uparrow s \end{pmatrix} \\
&\quad \begin{pmatrix} r \\ s \end{pmatrix} \otimes \begin{pmatrix} r' \\ s' \end{pmatrix} = \begin{pmatrix} r+r' \\ s \uparrow s' \end{pmatrix}
\end{aligned}$$

The function above is not in the form of parallelizable reduction, since there are extra calls of g for each subtree. To obtain a function in the form of parallelizable reduction, we fuse functions g and mcs' and introduce function $mcs2'$ defined as $mcs2'\ t = g\ (mcs'\ t)$ and operator \oplus' defined as $b \oplus' t = g\ (b \oplus t)$, that is, $b \oplus' (r, s)^\top = ((b+r) \uparrow 0, (b+r) \uparrow s)^\top$. For the top-level call of $mcs2'$, we select the second value with function snd . We then obtain the following definition, which is in the form of parallelizable reduction.

$$\begin{aligned}
mcs2\ t &= snd\ (mcs2'\ t) \\
mcs2'\ (RLeaf\ a) &= (a \uparrow 0, a \uparrow 0)^\top \\
mcs2'\ (RNode\ b\ [t_1, t_2, \dots, t_n]) &= b \oplus' (mcs2'\ t_1 \otimes mcs2'\ t_2 \otimes \dots \otimes mcs2'\ t_n)
\end{aligned}$$

Derive associative operator

The \otimes operator in the definition above is fortunately not only associative but also commutative, because of the associativity and commutativity of \uparrow and $+$. The unit of \otimes is $\iota_{\otimes} = (0, -\infty)^{\top}$.

Derive extended-distributive operator

To evaluate whether operator \otimes is extended-distributive over operator \oplus' , we match the following expression E_1 to E_2 , by simplifying them.

$$E_1 = a \oplus' \left(\left(\begin{pmatrix} r \\ s \end{pmatrix} \otimes \left(a' \oplus' \left(\begin{pmatrix} r' \\ s' \end{pmatrix} \otimes \begin{pmatrix} x_r \\ x_s \end{pmatrix} \right) \right) \right) \quad E_2 = A \oplus' \left(\begin{pmatrix} R \\ S \end{pmatrix} \otimes \begin{pmatrix} x_r \\ x_s \end{pmatrix} \right)$$

Due to space limitations, we will only provide the results of calculation.

$$E_1 = \left(\begin{array}{c} ((a + r + a' + r') + x_r) \uparrow ((a + r) \uparrow 0) \\ (((a + r + a' + r') \uparrow (a' + r')) + x_r) \uparrow ((a + r) \uparrow s \uparrow s') \uparrow x_s \end{array} \right)$$

$$E_2 = \left(\begin{array}{c} (A + R + x_r) \uparrow 0 \\ (A + R + x_r) \uparrow S \uparrow x_s \end{array} \right)$$

Operator \otimes is not extended-distributive since there are two conflicts in the calculation above. The first is that E_2 includes two $(A + R)$'s but the corresponding parts in E_1 have difference definitions. The other is that E_2 includes constant value 0 but the corresponding part in E_1 is not constant. To resolve these conflicts, we generalize the definition of \oplus' by assigning two variables a and b to the two occurrences of a respectively, and variable c for constant 0. The definition for generalized operator \oplus'' , its unit $\iota_{\oplus''}$, and the function that converts the left argument of \oplus' to that of \oplus'' are given as follows.

$$\begin{pmatrix} a \\ b \\ c \end{pmatrix} \oplus'' \begin{pmatrix} r \\ s \end{pmatrix} = \begin{pmatrix} (a + r) \uparrow c \\ (b + r) \uparrow s \end{pmatrix}, \quad \iota_{\oplus''} = \begin{pmatrix} 0 \\ -\infty \\ -\infty \end{pmatrix}, \quad conv \ a = \begin{pmatrix} a \\ a \\ 0 \end{pmatrix}$$

With operator \oplus'' and function $conv$, we can rewrite the definition of $mcs2'$ as:

$$mcs2' (RNode \ b \ [t_1, t_2, \dots, t_n]) = conv \ b \ \oplus'' (mcs2' \ t_1 \ \otimes \ mcs2' \ t_2 \ \otimes \ \dots \ \otimes \ mcs2' \ t_n).$$

Now, we can again evaluate whether \otimes is extended-distributive over the newly defined \oplus'' by simplifying the following two expressions, and finding the matches between them.

$$E_1 = \begin{pmatrix} a_1 \\ b_1 \\ c_1 \end{pmatrix} \oplus'' \left(\begin{pmatrix} r_1 \\ s_1 \end{pmatrix} \otimes \left(\begin{pmatrix} a_2 \\ b_2 \\ c_2 \end{pmatrix} \oplus'' \left(\begin{pmatrix} r_2 \\ s_2 \end{pmatrix} \otimes \begin{pmatrix} x_r \\ x_s \end{pmatrix} \right) \right) \right)$$

$$E_2 = \begin{pmatrix} A \\ B \\ C \end{pmatrix} \oplus'' \left(\begin{pmatrix} R \\ S \end{pmatrix} \otimes \begin{pmatrix} x_r \\ x_s \end{pmatrix} \right)$$

Now again, we only show the results of calculation.

$$E_1 = \left(\begin{array}{c} ((a_1 + r_1 + a_2 + r_2) + x_r) \uparrow ((a_1 + r_1 + c_2) \uparrow c_1) \\ (((b_1 + r_1 + a_2 + r_2) \uparrow (b_2 + r_2)) + x_r) \uparrow ((b_1 + r_1 + c_2) \uparrow s_1 \uparrow s_2) \uparrow x_s \end{array} \right)$$

$$E_2 = \left(\begin{array}{c} (A + R + x_r) \uparrow C \\ (B + R + x_r) \uparrow S \uparrow x_s \end{array} \right)$$

From these results, we obtain the following correspondences.

$$\begin{aligned} A + R &= a_1 + r_1 + a_2 + r_2 \\ B + R &= (b_1 + r_1 + a_2 + r_2) \uparrow (b_2 + r_2) \\ C &= (a_1 + r_1 + c_2) \uparrow c_1 \\ S &= (b_1 + r_1 + c_2) \uparrow s_1 \uparrow s_2 \end{aligned}$$

Since there are many solutions to the correspondences above, we obtain one solution by fixing R as 0, for example. We can then derive the following characteristic functions from the correspondences above.

$$\begin{aligned} p_1 \left(\begin{pmatrix} a_1 \\ b_1 \\ c_1 \end{pmatrix}, \begin{pmatrix} r_1 \\ s_1 \end{pmatrix}, \begin{pmatrix} a_2 \\ b_2 \\ c_2 \end{pmatrix}, \begin{pmatrix} r_2 \\ s_2 \end{pmatrix} \right) &= \begin{pmatrix} a_1 + r_1 + a_2 + r_2 \\ (b_1 + r_1 + a_2 + r_2) \uparrow (b_2 + r_2) \\ (a_1 + r_1 + c_2) \uparrow c_1 \end{pmatrix} \\ p_2 \left(\begin{pmatrix} a_1 \\ b_1 \\ c_1 \end{pmatrix}, \begin{pmatrix} r_1 \\ s_1 \end{pmatrix}, \begin{pmatrix} a_2 \\ b_2 \\ c_2 \end{pmatrix}, \begin{pmatrix} r_2 \\ s_2 \end{pmatrix} \right) &= \begin{pmatrix} 0 \\ (b_1 + r_1 + c_2) \uparrow s_1 \uparrow s_2 \end{pmatrix} \end{aligned}$$

Derive parallel program

Since we have verified the extended-distributivity of \otimes over \oplus'' and derived characteristic functions p_1 and p_2 in the previous step, we are ready to derive a parallel algorithm based on Corollary 3. Simply substituting the operators and functions for Corollary 3, we obtain the following parallel algorithm. Here, the two contracting operations, ϕ_L and ϕ_R , have the same definition, namely ϕ .

$$\begin{aligned} \psi_1 (\text{OrgLeaf } a) &= \begin{pmatrix} a \uparrow 0 \\ a \uparrow 0 \end{pmatrix} & \psi_2 (\text{OrgNode } b) &= \begin{pmatrix} \begin{pmatrix} b \\ b \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ -\infty \end{pmatrix} \end{pmatrix} \\ \psi_1 \text{OrgNil} &= \begin{pmatrix} 0 \\ -\infty \end{pmatrix} & \psi_2 \text{Dummy} &= \begin{pmatrix} \begin{pmatrix} 0 \\ -\infty \\ -\infty \end{pmatrix}, \begin{pmatrix} 0 \\ -\infty \end{pmatrix} \end{pmatrix} \\ \phi \left(\begin{pmatrix} a_1 \\ b_1 \\ c_1 \end{pmatrix}, \begin{pmatrix} r_1 \\ s_1 \end{pmatrix}, \begin{pmatrix} x_r \\ x_s \end{pmatrix}, \begin{pmatrix} a_2 \\ b_2 \\ c_2 \end{pmatrix}, \begin{pmatrix} r_2 \\ s_2 \end{pmatrix} \right) &= \begin{pmatrix} a_1 + r_1 + x_r + a_2 + r_2 \\ (b_1 + r_1 + x_r + a_2 + r_2) \uparrow (b_2 + r_2) \\ (a_1 + r_1 + x_r + c_2) \uparrow c_1 \end{pmatrix}, \begin{pmatrix} 0 \\ (b_1 + r_1 + x_r + c_2) \uparrow s_1 \uparrow x_s \uparrow s_2 \end{pmatrix} \\ G \left[\begin{pmatrix} \begin{pmatrix} a \\ b \\ c \end{pmatrix}, \begin{pmatrix} r \\ s \end{pmatrix} \end{pmatrix} \right] &= \lambda \begin{pmatrix} x_r \\ x_s \end{pmatrix} \begin{pmatrix} y_r \\ y_s \end{pmatrix} \cdot \begin{pmatrix} a \\ b \\ c \end{pmatrix} \oplus'' \left(\begin{pmatrix} r \\ s \end{pmatrix} \otimes \begin{pmatrix} x_r \\ x_s \end{pmatrix} \otimes \begin{pmatrix} y_r \\ y_s \end{pmatrix} \right) \end{aligned}$$

Observing the definitions of ψ_2 and ϕ above, we can find that r (the first value of the second tuple) returned by both ψ_2 and ϕ is always 0. It follows that we can remove variable r from the definition after substituting 0 for every occurrence of r , r_1 , and r_2 . Substituting 0 and simplifying the expressions, we successfully derive the following efficient parallel program.

$$\begin{aligned} \psi_1 (\text{OrgLeaf } a) &= \begin{pmatrix} a \uparrow 0 \\ a \uparrow 0 \end{pmatrix} & \psi_2 (\text{OrgNode } b) &= \begin{pmatrix} \begin{pmatrix} b \\ b \\ 0 \end{pmatrix}, -\infty \end{pmatrix} \\ \psi_1 \text{OrgNil} &= \begin{pmatrix} 0 \\ -\infty \end{pmatrix} & \psi_2 \text{Dummy} &= \begin{pmatrix} \begin{pmatrix} 0 \\ -\infty \\ -\infty \end{pmatrix}, -\infty \end{pmatrix} \end{aligned}$$

$$\begin{aligned}
& \phi \left(\left(\begin{pmatrix} a_1 \\ b_1 \\ c_1 \end{pmatrix}, s_1 \right) \begin{pmatrix} x_r \\ x_s \end{pmatrix} \left(\begin{pmatrix} a_2 \\ b_2 \\ c_2 \end{pmatrix}, s_2 \right) \right) \\
&= \left(\begin{pmatrix} a_1 + x_r + a_2 \\ (b_1 + x_r + a_2) \uparrow b_2 \\ (a_1 + x_r + c_2) \uparrow c_1 \end{pmatrix}, (b_1 + x_r + c_2) \uparrow s_1 \uparrow x_s \uparrow s_2 \right) \\
& G \left[\left(\begin{pmatrix} a \\ b \\ c \end{pmatrix}, s \right) \right] = \lambda \begin{pmatrix} x_r \\ x_s \end{pmatrix} \begin{pmatrix} y_r \\ y_s \end{pmatrix} \cdot \begin{pmatrix} a \\ b \\ c \end{pmatrix} \oplus'' \left(\begin{pmatrix} 0 \\ s \end{pmatrix} \otimes \begin{pmatrix} x_r \\ x_s \end{pmatrix} \otimes \begin{pmatrix} y_r \\ y_s \end{pmatrix} \right)
\end{aligned}$$

We know that we need four values in the parallel program for the maximum segment sum problem on lists [6, 10] and the maximum independent sum problem on binary trees [14]. The derived parallel program with our approach is reasonably efficient, since it also uses four values despite its applicability to trees with arbitrary shapes.

7 Conclusion

We developed a new methodology to systematically derive efficient parallel programs on trees with arbitrary shapes. Our methodology consists of three foundations: a new formalization of conditions for shunt contraction (Theorem 1), an extended-distributive property that replaces associativity and distributivity, and the parallelization of a class of reduction on rose trees (Theorem 2). The formalization of conditions for shunt contraction enables us to make a parallel program based on the tree contraction approach in a more constructive way. The extended-distributive property is very powerful since it can uniformly deal with associative operators, distributive operators, and other operators. Furthermore, we can find an extended-distributive operator more systematically by generalizing the definition and examining matching. The definition of parallelizable reduction is so practical that we can apply it to many programs.

The power of our method was demonstrated in the derivation of a parallel program for the maximum connected-sum problem. This problem first motivated us to develop the methodology, since we could not derive a parallel program for the problem using the techniques that have been proposed so far: operators \oplus and \otimes do not satisfy the distributive law although $+$ and \uparrow do, and the definition is not simple enough to enable us to parallelize it instinctively. In Section 6, we discussed how we systematically derived a parallel program, which is reasonably efficient. To the best of our knowledge, this is the first derivation of a parallel program for the maximum connected-sum problem.

We are currently working on generalizing this methodology so that we can deal with recursive datatypes more efficiently. In addition, we are working on applying the extended-distributive property to other situations: for example, fusing of successive calls of parallel skeletons on lists and deriving more general skeletons for nested lists.

References

1. K. Abrahamson, N. Dadoun, D. G. Kirkpatrick, and T. Przytycka. A simple parallel tree contraction algorithm. *Journal of Algorithms*, 10(2):287–302, June 1989.
2. J. Ahn and T. Han. An analytical method for parallelization of recursive functions. *Parallel Processing Letters*, 10(1):87–98, 2000.
3. J. Bentley. Column7: Algorithm design techniques. In *Programming Pearls*, pages 69–80. Addison-Wesley, 1986.
4. W.N. Chin, A. Takano, and Z. Hu. Parallelization via context preservation. *IEEE Computer Society International Conference on Computer Languages (ICCL'98)*, pages 153–162, May 1998.

5. M. Cole. *Algorithmic skeletons : a structured approach to the management of parallel computation*. Research Monographs in Parallel and Distributed Computing, Pitman, London, 1989.
6. M. Cole. Parallel programming, list homomorphisms and the maximum segment sum problems. Report CSR-25-93, Department of Computing Science, The University of Edinburgh, May 1993.
7. H. Deldari, J. R. Davy, and P. M. Dew. Parallel csg, skeletons and performance modelling. In *the Second Annual CSI Computer Conference (CSICC'96)*, pages 115–122, 1996.
8. J. Gibbons, W. Cai, and D. B. Skillicorn. Efficient parallel algorithms for tree accumulations. *Science of Computer Programming*, 23(1):1–18, 1994.
9. S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In *Annual European Conference on Parallel Processing, LNCS 1124*, pages 401–408, LIP, ENS Lyon, France, August 1996. Springer-Verlag.
10. Z. Hu, H. Iwasaki, and M. Takeichi. Construction of list homomorphisms by tupling and fusion. In *21st International Symposium on Mathematical Foundation of Computer Science, LNCS 1113*, pages 407–418, Cracow, September 1996. Springer-Verlag.
11. Z. Hu, H. Iwasaki, and M. Takeichi. Formal derivation of parallel program for 2-dimensional maximum segment sum problem. In *Annual European Conference on Parallel Processing, LNCS 1123*, pages 553–562, LIP, ENS Lyon, France, August 1996. Springer-Verlag.
12. Z. Hu, H. Iwasaki, and M. Takeichi. Formal derivation of efficient parallel programs by construction of list homomorphisms. *ACM Transactions on Programming Languages and Systems*, 19(3):444–461, 1997.
13. Z. Hu, M. Takeichi, and H. Iwasaki. Towards polytypic parallel programming. Technical Report METR 98-09, University of Tokyo, 1998.
14. K. Matsuzaki, Z. Hu, and M. Takeichi. Parallelization with tree skeletons. In *Annual European Conference on Parallel Processing (Euro-Par 2003), LNCS 2790*, pages 789–798, Klagenfurt, Austria, Aug 2003. Springer-Verlag.
15. G. L. Miller and J. H. Reif. Parallel tree contraction and its application. In *26th Annual Symposium on Foundations of Computer Science*, pages 478–489, Portland, OR, October 1985. IEEE Computer Society Press.
16. G. L. Miller and J. H. Reif. Parallel tree contraction part 2: Further applications. *SIAM Journal on Computing*, 20(6):1128–1147, December 1991.
17. F. Rabhi and S. Gorlatch. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer-Verlag New York Inc., 2002.
18. J. H. Reif and S. R. Tate. Dynamic parallel tree contraction. In *Proceedings of the Symposium on Parallel Algorithms and Architecture*, pages 114–121, 1994.
19. D. B. Skillicorn. The bird-meertens formalism as a parallel model. In J. S. Kowalik and L. Grandinetti, editors, *Software for Parallel Computation*, volume 106 of *NATO ASI Series F*, pages 120–133. Springer-Verlag, 1993.
20. D. B. Skillicorn. *Foundations of Parallel Programming*. Cambridge University Press, 1994.
21. D. B. Skillicorn. Parallel implementation of tree skeletons. *Journal of Parallel and Distributed Computing*, 39(2):115–125, 1996.
22. D. B. Skillicorn. A parallel tree difference algorithm. *Information Processing Letters*, 60(5):231–235, 1996.
23. D. B. Skillicorn. Structured parallel computation in structured documents. *Journal of Universal Computer Science*, 3(1):42–68, 1997.