IPSJ Transactions on Programming Vol.7 No.5 1–9 (Dec. 2014)

Regular Paper

Parallel Tree Contraction with Fewer Types of Primitive Contraction Operations and Its Application to Trees of Unbounded Degree

Akimasa Morihata[1,a)] Kiminori Matsuzaki[2,b)]

Received: June 22, 2014, Accepted: August 22, 2014

Abstract: Parallel tree contraction is a well established method of parallel tree processing. There are efficient and useful algorithms for binary trees, including the Shunt contraction algorithm and one based on the *m*-bridge decomposition method. However, for trees of unbounded degree, there are few practical tree contraction algorithms. The standard approach is "binarization," namely to translate the input tree to a full binary tree beforehand. To prevent the overhead introduced by binarization, we previously proposed the Rake-Shunt contraction algorithm (ICCS 2011), which is a generalization of the Shunt contraction algorithm to trees of unbounded degree. This paper further extends this result. The major contribution is to show that the Rake-Shunt contraction algorithm is a tree contraction algorithm that uses fewer types of primitive contraction operations if we assume the input tree has been binarized. This observation clarifies the connection between the Rake-Shunt contraction algorithm and those based on binarization. In particular, it enables us to translate a parallel program developed based on the Rake-Shunt contraction algorithm to one based on the *m*-bridge decomposition method. Thus, we can choose whether to use binarization according to the situation.

Keywords: parallel tree contraction, rose tree, *m*-bridge decomposition

1. Introduction

We investigate a method of parallel processing over tree-structured data. One of the simplest methods is processing independent subtrees in parallel. This method can finish in $O(\log N + H)$ parallel steps where N and H are the size and height of the tree-structured data, respectively. The computational complexity shows that this method is not effective for tall slender tree structures such as lists.

Parallel tree contraction [13] is a method of achieving a good parallel speedup regardless of the shape of the underlying tree structure. In parallel tree contraction, we process computation by gathering information scattered over the tree structure by using primitive contraction operations. Parallel tree contraction finishes in $O(\log N)$ parallel steps for a tree structure of any shape. Moreover, parallel tree contraction is a generalization of the simple subtree-based method if we combine it with a method called *m*-bridge decomposition [5], [7], [9], [17].

Parallel tree contraction algorithms for binary trees have been extensively studied; however, those for trees whose degree is unbounded (we call such trees *rose trees*) have been less studied. Most studies [1], [3], [4], [11], [12], [14] were based on *binarization*, namely, transforming the rose tree to a binary tree then applying a parallel tree contraction algorithm for binary trees. Binarization is effective in theory, since it only affects the constant factor — it at most doubles the number of nodes. Nevertheless, in practice, binarization may make it difficult to use parallel tree contraction. To use binarization, we should first translate our objective, namely an operation on rose trees, to that on binary trees, then implement it by using parallel tree contraction. Thus, we would like to avoid binarization if possible.

We previously proposed the *Rake-Shunt contraction algorithm* [15], which is a parallel tree contraction algorithm for rose trees. This algorithm finishes in $O(\log N)$ parallel steps, the same as other algorithms. Moreover, it does not use binarization; therefore, it prevents overheads and avoids difficulties possibly caused by binarization.

However, binarization is sometimes essential, mainly because of the following two reasons. First, rose trees are often implemented by the list-of-children structure, which is a binary tree. To adopt this implementation, parallel tree contraction algorithms for rose trees should be parallel tree contraction on binary trees as well. Second, the *m*-bridge decomposition method does not work well for rose trees. It makes all the siblings under a certain node as independent tasks, which may be too many and lead to too large overheads [17]. A natural solution to this problem is to binarize the rose tree; thereby, reducing the number of siblings [7], [12]. Therefore, we would like to unify parallel tree contraction algorithms for rose trees with those for binary trees.

We extend the Rake-Shunt contraction algorithm from our previous work [15] and address the above-mentioned issues. We show that the Rake-Shunt contraction algorithm is, even if we

[1] The University of Tokyo, Meguro, Tokyo 153–8902, Japan
[2] Kochi University of Technology, Kami-shi, Kochi 782–8502, Japan
[a)] morihata@graco.c.u-tokyo.ac.jp
[b)] matsuzaki.kiminori@kochi-tech.ac.jp

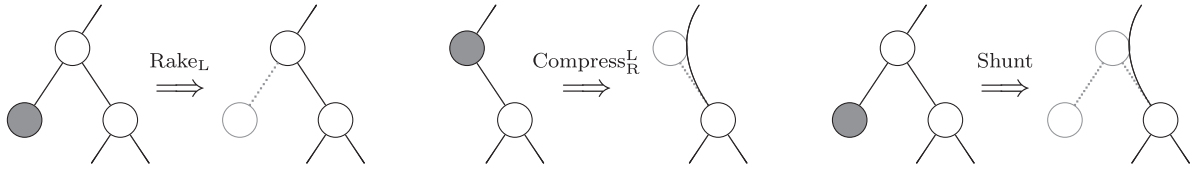

Fig. 1 Primitive contraction operations applied to gray nodes: Rake (left), Compress (middle), and Shunt (right).

regard it as one processing a binary, list-of-children representation of the rose tree, still a parallel tree contraction algorithm. Moreover it is nonstandard in the sense that it uses fewer types of primitive contraction operations. Classic parallel tree contraction algorithms use two types of primitive contraction operations: Rake and Compress. The Rake-Shunt contraction algorithm can be regarded as one that applies Compress operations only when a certain condition holds.

This observation leads to important consequences. First, the fact that the Rake-Shunt contraction algorithm consists of primitive contraction operations on binary trees implies that such operations on rose trees used in this algorithm are meaningful even on binary-tree representations. Thus, the primitive contraction operations on rose trees can be used even when we use other parallel tree contraction algorithms for binary trees via binarization. In particular, we argue that an implementation based on the Rake-Shunt contraction algorithm can be translated to one based on the *m*-bridge decomposition method. Therefore, we can chose appropriate implementations, for example whether we use binarization. Second, the relationship between primitive contraction operations for rose trees and those for binary trees is useful for theoretically analyzing the properties of parallel tree contraction algorithms for rose trees. For instance, the fact that the Rake-Shunt contraction algorithm uses fewer types of primitive contraction operations implies that it is at least no more difficult to implement our objective by using the algorithm than the approach based on binarization. Furthermore, since parallel tree contraction is a method for enabling O(log N)-time processing by allowing Compress operations, it is interesting that the same computational complexity can be achieved under a more restrictive setting.

The rest of this paper is organized as follows. Section 2 explains the results regarding parallel tree contraction including the Rake-Shunt contraction algorithm [15]. Section 3 shows the main result, that is, the Rake-Shunt contraction algorithm can be regarded as a parallel tree contraction algorithm on binary trees that uses fewer types of primitive contraction operations if we regard it as manipulating a binary tree that embodies the list-of-children representation. From this fact, we can translate tree processing based on the Rake-Shunt contraction algorithm to that based on the *m*-bridge decomposition method. Section 4 reports on the experiments carried out to confirm the effectiveness of our approach. Section 5 compares our results with related work, and Section 6 concludes the paper.

2. Parallel Tree Contraction

This section reviews the results regarding parallel tree contraction. Sections 2.3 and 2.4 introduce our previous results [15], which form the basis of this study. To make this paper self-contained, we show proofs for important theorems and lemmas.

2.1 Basic Definitions

We use the exclusive-read exclusive-write parallel random access machine (EREW PRAM) as our model for parallel computers. The numbers of the processors and of the nodes of the input tree are respectively denoted as P and N. We use functional programming language Haskell [16] for describing computations.

We discuss the following two types of tree structures. One is the binary tree, which can be expressed as an algebraic data structure in Haskell as follows.

$$\textbf{data}\ \mathsf{BTree}_{A,B} = \mathsf{Tip}\ A \mid \mathsf{Bin}\ \mathsf{BTree}_{A,B}\ B\ \mathsf{BTree}_{A,B}$$

The other is the rose tree defined as follows.

$$\textbf{data}\ \mathsf{RTree}_{A} = \mathsf{Node}\ A\ [\mathsf{RTree}_{A}]$$

We assume that both structures are implemented using arrays of nodes each of which has bidirectional pointers between the parent and the child. Note that in a rose tree, siblings are ordered and connected accordingly.

We call transforming rose trees to binary trees *binarization*, which is expressed by the following *binarize* function.

```
binarize :: RTree_A → BTree_A,()
binarize rt = aux [rt]
  where
    aux [ ]                 =  Tip ()
    aux (Node a cs : rts)   =  Bin (aux cs) a (aux rts)
```

For each node in the rose tree, its children and the following siblings respectively become the left and right subtrees. Though other binarization methods have been considered, e.g., Ref. [7], we consider *binarize* only.

2.2 Primitive Contraction Operations

We now briefly introduce parallel tree contraction. Details can be found in a book by Reif [17].

Parallel tree contraction collapses a tree structure into a node by using primitive contraction operations. The following are the three standard primitive contraction operations.

Rake Remove a leaf.

Compress Remove an internal node that has exactly one child.

Shunt For a leaf that has exactly one sibling, remove it and its parent.

Figure 1 illustrates these operations. It is worth noting that a Shunt operation is a successive application of the Rake and Compress operations.

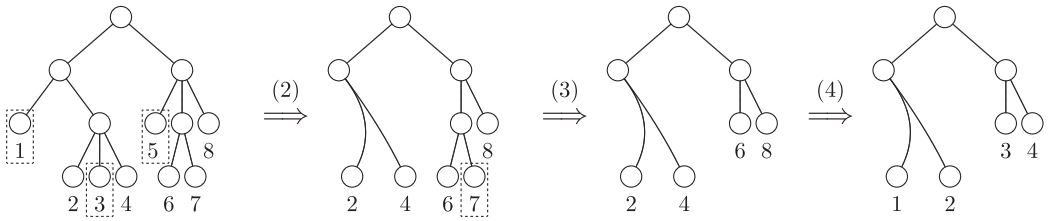

Fig. 2 Outline of Procedure 1: the dashed-lined boxes show leaves to which primitive contraction operations are to be applied.

2.3 Rake-Shunt Contraction Algorithm

The Rake-Shunt contraction algorithm we previously proposed [15] is a parallel tree contraction algorithm for rose trees. It is a generalization of the classic Shunt contraction algorithm [1], [8], [17] for binary trees. They are identical for binary trees, and moreover, the former can be used for non-binary trees as well. For simplicity, we assume that the input tree does not contain any nodes that have exactly one child. We can fulfill this assumption by inserting dummy nodes that have no effect on computation. We also previously proposed another method that avoids using dummy nodes [15].

Procedure 1 (Rake-Shunt Contraction Algorithm)**.**

(1) Number leaves from left to right, starting from 1.
(2) Remove all odd-numbered leaves each of which is not the rightmost sibling as follows: if the leaf has exactly one sibling, apply Shunt to it; otherwise, apply Rake to it.
(3) Remove all odd-numbered leaves (which should be the rightmost) as follows: if the leaf has exactly one sibling, apply Shunt to it; otherwise, apply Rake to it.
(4) Halve all the numbers.
(5) If the tree consists of more than one node, go back to Step (2). □

Figure 2 illustrates Procedure 1.

We should be careful regarding implementation of Shunt operations so that we can simultaneously apply all the primitive contraction operations in the same step. Our implementation of a Shunt operation removes a leaf and its sibling, not its parent. The resulting trees are equivalent. This implementation keeps sibling relations; therefore, it is better suited for the list-of-children implementation of rose trees. More discussions can be found in our previous paper [15].

If we would like to implement tree processing based on parallel tree contraction algorithms, we merge the information retained by the removed nodes with that of their neighbors. We explain this in detail later. In the Rake-Shunt contraction algorithm, we transfer the information to the right sibling for each Rake operation in Step (2), to the left sibling for each Rake operation in Step (3), and to the parent for each Shunt operation.

Procedure 1 is efficient.

Lemma 2 (Ref. [15])**.** In Procedure 1, in each step, no two primitive contraction operations touch the same node.

Proof. The proof is almost the same as with the Shunt contraction algorithm [1], [8], [17]. □

Lemma 3 (Ref. [15])**.** Procedure 1 finishes in time $O(N/P + \log P)$.

Proof. It is nearly the same as with the Shunt contraction algorithm [1], [8], [17]. Procedure 1 removes all the odd-numbered leaves for each iteration; thus, $\lceil \log N \rceil$ iterations are sufficient. The total amount of work is apparently $O(N)$. Now, the computational complexity follows from Brent's scheduling lemma [2]. □

2.4 Tree Processing by Parallel Tree Contraction

Parallel tree contraction algorithms show a schedule to process tree structures in parallel. Thus, if our objective can be decomposed into local computations each of which corresponds to a primitive contraction operation, the objective can be efficiently achieved by following the schedule.

For example, consider calculating the total sum of the values in a tree. We can implement this calculation by adding the values of the removed node to the neighbor for each Rake/Compress operation. Hence, by using Procedure 1, we can calculate the total sum in $O(\log N)$ parallel steps.

We can implement more complicated operations. We consider tree reductions [11], [12], [18]. Given an associative binary operator, $(\oplus) :: A \to A \to A$, and a (possibly non-associative) binary operator, $(\otimes) :: B \to A \to A$, a tree reduction, $reduce_{\otimes,\oplus} :: \mathsf{RTree}_B \to A$, is identified as follows.

$$
\begin{aligned}
&reduce_{\otimes,\oplus}\ (\mathsf{Node}\ a\ [\,]) \\
&\qquad = \quad a \\
&reduce_{\otimes,\oplus}\ (\mathsf{Node}\ a\ [t_1, \ldots, t_n]) \\
&\qquad = \quad a \otimes (reduce_{\otimes,\oplus}\ t_1 \oplus \cdots \oplus reduce_{\otimes,\oplus}\ t_n)
\end{aligned}
$$

The following theorem shows that $reduce_{\otimes,\oplus}$ can be implemented using the Rake-Shunt contraction algorithm if we additionally supply a function, $\rho_{\otimes,\oplus} :: (A, B, A) \to (A, B, A) \to (A, B, A)$, that satisfies a certain property.

Theorem 4 (Theorem 3 of Ref. [15])**.** We can calculate $reduce_{\otimes,\oplus}$ in time $O(N/P + \log P)$ if $\oplus$ is associative, $\otimes$ and $\oplus$ are constant-time, and there exists a constant-time function, $\rho_{\otimes,\oplus}$, that satisfies the following property.

$$
\begin{aligned}
&\rho_{\otimes,\oplus}\ (a, b, c)\ (v, w, z) = (a', b', c') \\
&\quad \Longleftrightarrow\ (\lambda x \to a \oplus (b \otimes x) \oplus c) \circ (\lambda x \to v \oplus (w \otimes x) \oplus z) \\
&\qquad\qquad = (\lambda x \to a' \oplus (b' \otimes x) \oplus c')
\end{aligned}
$$

Proof. The key is that each internal node retains a closure of the form of $\lambda x \to p \oplus (q \otimes x) \oplus r$ ($p \in A$, $q \in B$, and $r \in A$ are values). In what follows, we denote this closure by a triple, $\langle p, q, r \rangle$.

In the initialization step, for each node having value a, we associate $\langle \iota_\oplus, a, \iota_\oplus \rangle$, where $\iota_\oplus$ is the unit of $\oplus$. Even if $\oplus$ does not have a unit, we can introduce a unit by using a special flag that

stands for the unit.

Then, for each primitive contraction operation, we process the computation as follows.

Rake Let a be the value of the leaf removed.

if its sibling is a leaf: Overwrite the value b of the sibling by $b \oplus a$ if the sibling is on its left and by $a \oplus b$ if on its right.

if its sibling is an internal node: Overwrite the value $\langle v, w, z\rangle$ of the sibling by $\langle v, w, z \oplus a\rangle$ if the sibling is on its left and by $\langle a \oplus v, w, z\rangle$ if on its right.

if it has no sibling: Overwrite the value $\langle v, w, z\rangle$ of its parent by $v \oplus (w \otimes a) \oplus z$.

Compress Let $\langle a, b, c\rangle$ be the value of the removed internal node.

if its child is a leaf: Overwrite the value v of the child by $a \oplus (b \otimes v) \oplus c$.

if its child is an internal node: Overwrite the value $\langle v, w, z\rangle$ of the child by $\langle a', b', c'\rangle$ where $\rho_{\otimes,\oplus}\ (a, b, c)\ (v, w, z) = (a', b', c')$.

It is not difficult to see that the above procedure correctly computes the tree reduction. □

As an example, we consider calculating the maximum total sum of values in a subtree. The following function, *mss*, calculates that value, where *maximum* extracts the maximum from a set.

$$
\begin{aligned}
&mss\ (\mathsf{Node}\ a\ ts)\\
&\quad = \quad \textbf{let}\ [(m_1, s_1), \ldots, (m_n, s_n)] = map\ mss\ ts\\
&\qquad\qquad s' = a + \textstyle\sum_{1 \le i \le n} s_i\\
&\qquad\ \textbf{in}\ (maximum\ \{m_1, \ldots, m_n, s'\}, s')
\end{aligned}
$$

We can implement it by $reduce_{\otimes,\oplus}$, where $\otimes$ and $\oplus$ are specified as follows.

$$
\begin{aligned}
&a \otimes (m, s) = (max\ m\ (a + s), a + s)\\
&(m_1, s_1) \oplus (m_2, s_2) = (max\ m_1\ m_2, s_1 + s_2)
\end{aligned}
$$

Note that $\oplus$ is associative. Therefore, we can use Theorem 4 by providing $\rho_{\otimes,\oplus}$. Since $\oplus$ is commutative, it is sufficient to consider closures of a simpler form, $\lambda x \to a \oplus (b \otimes x)$. Let $\uparrow$ be the binary maximum operator, which binds looser than $+$.

$$
\begin{aligned}
&(\lambda x \to (a_m, a_s) \oplus (b \otimes x)) \circ (\lambda x \to (v_m, v_s) \oplus (w \otimes x))\\
&= \ \{\ \text{by definitions of } \otimes \text{ and } \oplus\ \}\\
&\quad \lambda(x_m, x_s) \to \textbf{let}\ s' = w + x_s;\ \ s'' = b + v_s + s'\\
&\qquad\qquad\qquad \textbf{in}\ (b_m \uparrow v_m \uparrow x_m \uparrow s' \uparrow s'', a_s + s'')\\
&= \ \{\ \text{by definitions of } \otimes \text{ and } \oplus\ \}\\
&\quad \textbf{let}\ a' = (b_m \uparrow v_m, a_s + b + v_s + w - b')\\
&\qquad b' = w \uparrow b + v_s + w\\
&\quad \textbf{in}\ (\lambda x \to a' \oplus (b' \otimes x))
\end{aligned}
$$

Therefore, the following $\rho_{\otimes,\oplus}$ satisfies our requirement.

$$
\begin{aligned}
&\rho_{\otimes,\oplus}\ ((a_m, a_s), b, \_)\ ((v_m, v_s), w, \_)\\
&\quad = \textbf{let}\ b' = w \uparrow b + v_s + w\\
&\qquad \textbf{in}\ ((b_m \uparrow v_m, a_s + b + v_s + w - b'), b', \_)
\end{aligned}
$$

Now Theorem 4 leads to an implementation based on parallel tree contraction.

The strong point of the Rake-Shunt contraction algorithm and Theorem 4 is that the approach avoids binarization. Thus, we can intuitively understand the role of each operator: $\oplus$ merges information of siblings, $\otimes$ merges information of a parent and its child, and $\rho_{\otimes,\oplus}$ is used for implementing Compress operations. In particular, this intuitive understanding makes it easier to develop $\rho_{\otimes,\oplus}$ from $\oplus$ and $\otimes$.

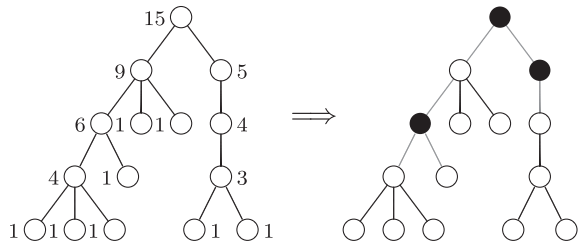

Fig. 3 m-bridge ($m = 5$): black nodes on right are 5-critical; nodes connected by solid lines form 5-bridge.

Matsuzaki et al. [11], [12] proposed a theorem similar to Theorem 4. Their theorem also requires a function similar to $\rho_{\otimes,\oplus}$ for implementing tree reduction by parallel tree contraction. The significant difference is that since their method is based on binarization, the required function is less intuitive: it essentially corresponds to a Compress operation on binary trees obtained by binarization. We see in Section 3.2 that the required function is at least not simpler if we use binarization.

2.5 *m*-bridge Decomposition

For each step, the Rake-Shunt contraction algorithm essentially manipulates all the nodes. This low locality is not only worse cache behavior but also requires a synchronization barrier between every step. The m-bridge decomposition method [5], [7], [9], [17] is another approach to parallel tree contraction. It uses divide-and-conquer: it decomposes a tree into substructures so that the substructures can be collapsed in parallel. Therefore, it reduces the number of synchronizations, improves locality, and moreover, enables us to use distributed parallel computers.

Let m be a natural number ($1 < m \le \mathrm{N}$). The root node of subtree $t = \mathsf{Node}\ a\ ts$ ($ts \ne [\,]$) is said to be *m-critical* if $\lfloor size(t)/m \rfloor > \max_{t' \in ts} \lfloor size(t')/m \rfloor$, where $size(t)$ denotes the number of the nodes in t. An *m-bridge* is a maximal set of connected nodes that contains no m-critical node. **Figure 3** shows 5-critical nodes and 5-bridges.

Apparently, each m-bridge consists of at most $m - 1$ nodes. Moreover, it is not difficult to see that a tree contains at most $2\mathrm{N}/m - 1$ m-critical nodes [17]. Therefore, by choosing appropriate m ($m = \mathrm{N}/\mathrm{P}$, for example), the m-bridge decomposition method yields a reasonable number of m-bridges of reasonable sizes.

Furthermore, each m-bridge can be fully collapsed by using primitive contraction operations; therefore, if our objective can be decomposed into computations that correspond to primitive contraction operations, the m-bridge decomposition method leads to divide-and-conquer parallel processing.

It is easy to collapse an m-bridge all of whose nodes do not have any m-critical children. We can collapse it by using Rake operations in a bottom-up manner.

The interesting case is that a node in the bridge has an m-

critical child. Note that such an m-critical child is unique because the least common ancestor of two m-critical nodes is m-critical. Thus, the only problematic part is the path from the root of the bridge to the node whose child is m-critical; other parts can be collapsed by Rake operations in a bottom-up manner. Once the other parts are removed, the path can be collapsed by using a series of Compress operations.

3. Decomposing Rake-Shunt Contraction to Primitive Contraction Operations and Its Application to m-bridge Decomposition

We have introduced the Rake-Shunt contraction algorithm, which is a generalization of a tree contraction algorithm for binary trees, namely the Shunt contraction algorithm. It avoids binarization; therefore, it makes it easier to derive operators for parallel processing over rose trees, namely $\rho_{\otimes,\oplus}$. However, the following issues remain.

- While the Rake-Shunt contraction algorithm avoids binarization for deriving $\rho_{\otimes,\oplus}$, it in fact manipulates binary tree representations, namely the list-of-children representations, for rose trees. Therefore, it is natural to expect that the Rake-Shunt contraction algorithm is a parallel tree contraction even on the binary tree representation. Nevertheless, its characterization as a tree contraction for binary trees is still unclear.
- It is natural to expect, as with the Rake-Shunt contraction algorithm, to generalize other methods of parallel tree contraction for binary trees including the m-bridge decomposition method. However, it is essentially difficult to develop an efficient parallel tree contraction algorithm for rose trees based on the m-bridge decomposition method. Since every child of a critical node yields an m-bridge, the m-bridge decomposition method applied to a rose tree may result in too many m-bridges. To overcome this problem, the current implementations apply binarization before performing m-bridge decomposition [7], [12]. Nevertheless, we would like to reduce overheads introduced by binarization.

In this section, we examine the first issue then address the second based on the answer to the first.

3.1 Decomposing Rake-Shunt Contraction Algorithm to Primitive Contraction Operations

First, we characterize the Rake-Shunt contraction algorithm as a manipulation on a binary tree representation. For this purpose, we identify the primitive contraction operations for rose trees appearing in the Rake-Shunt contraction algorithm as primitive contraction operations for binary trees. We classify the primitive contraction operations for binary trees as follows, which will be useful to characterize the Rake-Shunt contraction algorithm. $\mathrm{Rake}_{\mathrm{L}}$ and $\mathrm{Rake}_{\mathrm{R}}$ eliminate the left and right leaves, respectively. $\mathrm{Compress}^{\mathrm{L}}_{\mathrm{R}}$ removes an internal node that is a left child and has its right child. $\mathrm{Compress}^{\mathrm{L}}_{\mathrm{L}}$, $\mathrm{Compress}^{\mathrm{R}}_{\mathrm{L}}$, and $\mathrm{Compress}^{\mathrm{R}}_{\mathrm{R}}$ are similarly defined. Figure 1 shows $\mathrm{Rake}_{\mathrm{L}}$ and $\mathrm{Compress}^{\mathrm{L}}_{\mathrm{R}}$. We ignore Shunt operations because they can be decomposed into Rake and Compress operations. The root node can be regarded as either a left or a right child; the difference does not matter.

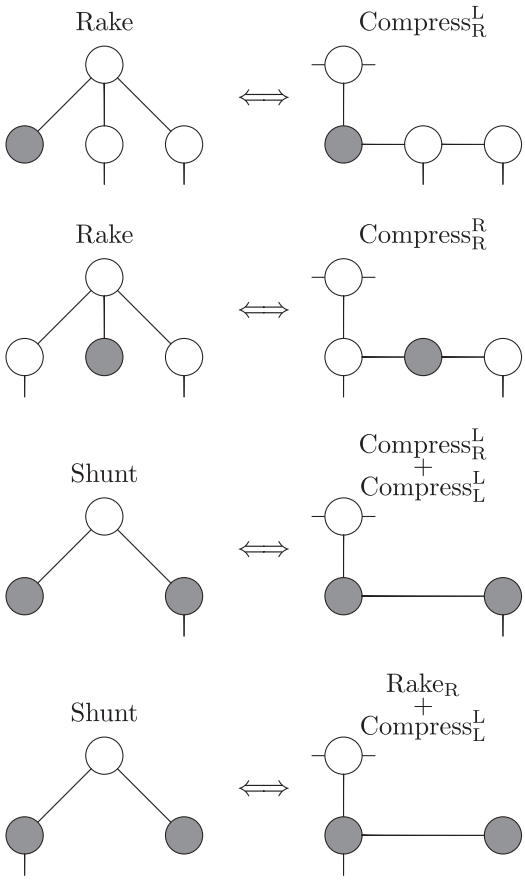

Fig. 4 Correspondence between primitive contraction operations on rose trees and those on binary trees: Left and right figures respectively show rose trees and binary trees; gray nodes are to be removed; on binary tress, we omit nodes that have no value.

We examine to what types of primitive contraction operations on binary trees Rake and Compress operations on rose trees correspond via binarization. **Figure 4** shows this correspondence. Roughly speaking, Rake operations on rose trees correspond to $\mathrm{Compress}^{\mathrm{L}}_{\mathrm{R}}$ or $\mathrm{Compress}^{\mathrm{R}}_{\mathrm{R}}$ on binary trees, Compress operations occurring at Shunt operations on rose trees correspond to $\mathrm{Compress}^{\mathrm{L}}_{\mathrm{L}}$. $\mathrm{Rake}_{\mathrm{L}}$ and $\mathrm{Rake}_{\mathrm{R}}$ occur when we remove a leaf that has no sibling and a leaf that is the last sibling, respectively, on rose trees. It is worth noting that at a Shunt operation in the Rake-Shunt contraction algorithm, we remove a leaf with its parent, not its sibling. This modification is essential; otherwise, we cannot find such a complete correspondence between the primitive contraction operations on rose trees and those on binary trees.

The observation so far has clarified that no primitive contraction operation on rose trees corresponds to $\mathrm{Compress}^{\mathrm{R}}_{\mathrm{L}}$ on binary trees. Hence, the Rake-Shunt contraction algorithm can be regarded as a procedure that collapses a binary tree in $\mathrm{O}(\log \mathrm{N})$ parallel steps without using $\mathrm{Compress}^{\mathrm{R}}_{\mathrm{L}}$.

We will discuss consequences of this fact in the next subsection. In the remainder of this subsection, we discuss whether efficient parallel tree contraction for binary trees is possible with fewer types of primitive contraction operations.

Apparently, $\mathrm{Rake}_{\mathrm{R}}$ and $\mathrm{Rake}_{\mathrm{L}}$ are necessary. Without them, we cannot eliminate leaves. $\mathrm{Compress}^{\mathrm{L}}_{\mathrm{L}}$ and $\mathrm{Compress}^{\mathrm{R}}_{\mathrm{R}}$ are also essential. For instance, without $\mathrm{Compress}^{\mathrm{L}}_{\mathrm{L}}$, we need $\mathrm{O}(\mathrm{N})$ parallel

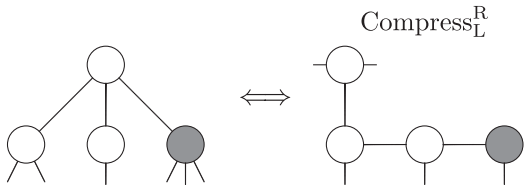

Fig. 5 Operation that corresponds to Compress$^{R}_{L}$ on binary tree: left and right figures respectively show rose trees and binary trees; gray nodes are to be removed; on binary tree, we omit nodes that have no value.

steps for collapsing a monadic tree each of whose internal node has only the left child.

Lemma 5. Rake$_{R}$, Rake$_{L}$, Compress$^{L}_{L}$, and Compress$^{R}_{R}$ are necessary for collapsing a binary tree in O(log N) parallel steps by using Rake and Compress operations. □

As shown in the discussion about the Rake-Shunt contraction algorithm, either of Compress$^{L}_{R}$ or Compress$^{R}_{L}$ is sufficient for efficient parallel tree contraction. Intuitively, we can understand this fact as follows. Assume that we cannot use Compress$^{L}_{R}$; the discussion about Compress$^{L}_{L}$ indicates that the possibly problematic case is that the input tree forms a path alternately growing as left-right-left-right-$\cdots$. This type of tree can be mostly collapsed using Compress$^{R}_{L}$. Note that this discussion also clarifies that either Compress$^{L}_{R}$ or Compress$^{R}_{L}$ is necessary for efficient parallel tree contraction.

Lemma 6. Either Compress$^{L}_{R}$ or Compress$^{R}_{L}$ is necessary for collapsing a binary tree in O(log N) parallel steps by using Rake and Compress operations. □

Lemma 7. By using Rake$_{R}$, Rake$_{L}$, Compress$^{L}_{L}$, Compress$^{R}_{R}$, and Compress$^{L}_{R}$, we can collapse a binary tree in O(log N) parallel steps. □

Corollary 8. By using Rake$_{R}$, Rake$_{L}$, Compress$^{L}_{L}$, Compress$^{R}_{R}$, and Compress$^{R}_{L}$, we can collapse a binary tree in O(log N) parallel steps. □

3.2 Properties of Parallel Tree Contraction without Compress$^{R}_{L}$

It is worthwhile to avoid using Compress$^{R}_{L}$. **Figure 5** shows a Compress$^{R}_{L}$ on a binary tree representation and the corresponding operation on a rose tree. The corresponding operation on a rose tree eliminates the rightmost sibling and makes its children siblings of it. Apparently, it is less intuitive than the primitive contraction operations on rose trees. Recall that we should provide the corresponding calculation for each primitive contraction operation so as to implement tree processing by parallel tree contraction. If we use Compress$^{R}_{L}$, we should work out a calculation for it; however, since it is not intuitive, it is rather difficult to work out one. At least, absence of Compress$^{R}_{L}$ makes it no more complicated to work out the corresponding calculations. This is the reason Theorem 4 is easier to use than that by Matsuzaki et al. [11], [12].

Nevertheless, the absence of Compress$^{R}_{L}$ is not very significant. As shown in the discussions in the previous subsection, the effect of the absence of Compress$^{R}_{L}$ is to delay the contraction process for only a single node. Thus, we can translate parallel tree processing without Compress$^{R}_{L}$ to those with Compress$^{R}_{L}$ by introducing function closures that express the delay. In summary, it is asymptotically equivalent whether we use Compress$^{R}_{L}$.

3.3 Parallel Computation Using *m*-bridge Decomposition without Compress$^{R}_{L}$

We have observed that the Rake-Shunt contraction algorithm is nothing but a parallel tree contraction algorithm without using Compress$^{R}_{L}$ on binary tree representations. This observation suggests that Compress$^{R}_{L}$ is essentially unnecessary for parallel tree contraction of binary trees. Thus, we can also consider Compress$^{R}_{L}$-free variants of other parallel tree contraction algorithms.

In what follows, we consider a Compress$^{R}_{L}$-free *m*-bridge decomposition method for binary trees, which leads to parallel tree processing of rose trees based on the *m*-bridge decomposition. As we have discussed, to avoid having many *m*-bridges, it is reasonable to binarize the rose tree and apply *m*-bridge decomposition. As we will show below, obtained *m*-bridges can be collapsed without using Compress$^{R}_{L}$. Therefore, by using $\oplus$, $\otimes$, and $\rho_{\oplus,\otimes}$ for the Rake-Shunt contraction algorithm, we can implement this collapsing process for parallel tree reductions. This approach has less synchronization and better locality than the Rake-Shunt contraction algorithm.

As mentioned in Section 2.5, in parallel tree contraction based on the *m*-bridge decomposition method, Compress operations are necessary only for eliminating the nodes on the path from the root of the bridge to the *m*-critical node. Other nodes can be easily eliminated by Rake operations in a bottom-up manner. Thus, we concentrate our effort on that path. Let c be the *m*-critical node, p be its parent, and g be its grandparent. Assume that g is not the root. Now, without using Compress$^{R}_{L}$, we can eliminate either p or g. If p is g's right child and c is p's left child, we cannot eliminate p. Then, for any cases, we can remove g by applying either Compress$^{L}_{R}$ or Compress$^{R}_{R}$. Otherwise, we can remove p. Therefore, by a series of compress operations, we can collapse each *m*-bridge into a structure that consists of at most one nodes.

4. Experiments

We report our experiments to examine the effectiveness of our approach.

The environment of our experiment consisted of an Intel Xeon E5-2640 2.5 GHz CPU (6 cores), 12 GB memory, g++ 4.8.2, Linux 3.13.0-29 (64-bit Ubuntu 14.04). Hyperthreading was disabled by the BIOS configuration.

We prepared two programs that calculate the maximum subtree sum, *mss*, discussed in Section 2.4; `seq` is a bottom-up sequential program and `parallel` is a parallel program based on the *m*-bridge decomposition method implemented using OpenMP, where m is an important tuning parameter, and we chose 2^{21} based on preliminary experiments. Both `seq` and `parallel` do not implement *reduce* but are specialized to *mss*; moreover, both are loop-based implementations that explicitly use stacks that retain parameters and return values.

The following four rose trees were used as the inputs.

bin: whose binary tree representation is a complete binary tree.

flat: all of whose nodes except the root are leaves; note that its

Table 1 Experimental results (units: milliseconds).

	seq	parallel					
		P=1	P=2	P=3	P=4	P=5	P=6
bin	2,060	2,121	1,193	770	608	496	431
flat	1,560	1,536	811	550	443	394	367
mono	1,448	1,557	834	568	451	396	370
lr	1,445	1,539	830	563	460	403	365

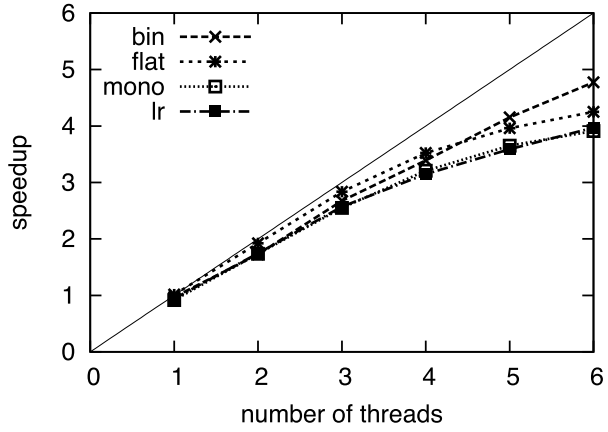

Fig. 6 Parallel speedup of `parallel` over `seq`.

binary tree representation is a monadic tree that grows right.

mono: all of whose internal nodes are monadic; note that its binary tree representation is a monadic tree that grows left.

lr: each of whose internal nodes has exactly two children and the left child is a leaf; note that its binary tree representation is a monadic tree that grows alternately left or right.

Each tree consists of $2^{27} - 1$ nodes. Each node has an integer ranging over −63 to 64.

The rose trees were implemented as follows. A rose tree consists of a set of *m*-bridges and a tree that represents adjacent relations between the bridges. Note that `seq` takes a tree that consists of exactly one bridge. Every bridge or tree is an array of nodes. Each node has its value, a pointer to its first child, and a pointer to its next sibling. Each pointer is represented by an index (i.e., an integer) of the array. The structure that represents a tree is directly generated into the memory at the initialization step of the program.

For each of the four inputs, we measured the execution times of `seq` and `parallel` while varying the number of threads, P. **Table 1** summarizes the results. **Figure 6** shows relative parallel speedups over the execution time of `seq`. Each execution time is an average of ten executions and does not include times for data generation.

First, compared with `seq`, `parallel` had a little overhead. The differences varied in the shape of trees and were less than ten percent. `parallel` became faster as more threads were available. It was four to five times faster than `seq` when six threads were available. Moreover, the performance was not seriously affected by the shape of the input tree. These results are ideal for parallel computing methods and imply that our approach is promising.

The current experiment is still preliminary. Different results might be obtained if we consider other tree processing than the maximum subtree sum, other types of input trees, and use other computational environments. Further experiments are for future work.

5. Related Work

Parallel tree contraction, proposed by Miller and Reif [13], is a method of efficient parallel processing of trees of any shape. Below we review the results that relate to ours. More information about classic results can be found in the book by Reif [17].

Miller and Reif [13] observed that even trees that require $O(N)$ parallel steps by bottom-up processing, i.e., by Rake operations, can be processed in $O(\log N)$ parallel steps if Compress operations are available. Based on this observation, they proposed a parallel tree contraction algorithm that consists of Rake and Compress operations. The algorithm has two shortcomings. It requires concurrent-read concurrent-write (CRCW) PRAM, which is unrealistic. Its amount of work is $O(N \log N)$; thus, not optimal. To resolve these problems, two methods were proposed. One is the Shunt contraction algorithm [1], [8], which uses Shunt operations rather than Compress operations. The other uses the *m*-bridge decomposition method [5]. Both algorithms run on EREW PRAM in time $O(N/P + \log N)$, which is optimal.

Binary trees are the major subject in the literature of parallel tree contraction. The Shunt contraction algorithm requires the input to be binary. The *m*-bridge decomposition method can be applied to non-binary trees but may result in too many *m*-bridges; therefore, its asymptotic complexity is worse.

These problems are addressed in the book by Reif [17]. It also showed methods for processing non-binary trees without decreasing the asymptotic complexity.

One such method is called *isolation*, which is a parallel tree contraction algorithm for rose trees. It delays, for each consecutive sequence of Compress-able nodes, removing the head node of the sequence until all the other nodes are removed. Thus, we should carefully determine whether each node is a head of such a sequence. Accordingly, its constant factor is large and it is less practical. To the best of our knowledge, its implementation and experiments have not been reported.

In addition, for the *m*-bridge decomposition method, the following improvement was addressed. For each *m*-critical node that has many children, apply parallel list ranking to determine the size of each bridge then group the bridges into reasonably-sized chunks. If we apply this method, we would like to fully collapse each chunk in parallel by using an operation to merge siblings. When such an operation is available, this method is essentially equivalent to the method based on binarization, i.e., binarizing the rose tree then applying the *m*-bridge decomposition method; rather, it is less efficient because of the cost of performing additional parallel list ranking. We are not aware of its implementation and experiments.

A more practical solution is binarization. In fact, most classic studies [1], [3], [4], [14] used binarization to avoid the above-mentioned problems. A notable work is that by Matsuzaki et al. [12], who reported on an implementation of typical parallel rose-tree processing patterns, called parallel (rose-) tree skeletons, based on parallel tree contraction. Their strategy is that they binarize rose trees then implement rose-tree processing by tree skeletons for binary trees, which are implemented based on the *m*-bridge decomposition method [10].

Issues concerning binarization include the method of translating rose-tree processing to binary tree processing and the method of parallelizing the obtained binary tree processing. However, most studies ignored these issues. A notable exception is that by Matsuzaki et al. [11], [12]. They showed that parallel rose-tree reductions can be translated to parallel binary-tree processing if the premise of Theorem 4 is fulfilled. However, the translation is complex. It may introduce significant overhead; moreover, it seems difficult to consider generalizing their results.

Previously, we [15] proposed the Rake-Shunt contraction algorithm, which is a parallel tree contraction algorithm for rose trees. This algorithm has two major strengths. Its behavior is a natural generalization of the Shunt contraction algorithm and simpler than the isolation-based method. It avoids binarization; thus, it is easier to discuss what types of parallel operations can be implemented based on it.

Based on these preceding studies, we showed that tree processing based on the Rake-Shunt contraction algorithm can be translated to one based on the *m*-bridge decomposition method. The major two methods for parallel tree contraction for binary trees have been generalized to implement rose tree operations. Moreover, we observed that this generalization was guided by the fact that the Rake-Shunt contraction algorithm is essentially a parallel tree contraction algorithm without $\mathrm{Compress}_{\mathrm{L}}^{\mathrm{R}}$ on binary tree representations. This observation is useful for understanding the properties of the Rake-Shunt contraction algorithm and the *m*-bridge decomposition method based on it. Furthermore, as a purely theoretical result of parallel tree contraction, it is interesting in its own right that $\mathrm{O}(\log \mathrm{N})$-step parallel tree contraction is possible with a more restrictive setting than what discussed in the literature.

Kawamura and Matsuzaki [7] discussed *m*-bridge decomposition of rose trees via binarization and proposed a binarization method especially suitable for processing large XML-like data by using MapReduce-like distributed memory parallel computing environments. Their binarization is different in the sense that the order of sibling is reversed: more right siblings in a rose tree are nearer to the root in the corresponding binary tree representation. It is easy to see that our result is applicable to this binarization as well.

Kakehi et al. [6] proposed a parallel tree contraction algorithm for rose trees, which is different from both the Shunt contraction algorithm and the *m*-bridge decomposition method. With their approach, it is assumed that the input is in an XML-like format. It treats the input as a sequence of tags, divides the input without caring the underlying tree structure, and processes each part in parallel. They showed that under the same condition as Theorem 4, their algorithm can calculate the rose tree reduction in time $\mathrm{O}(\mathrm{N}/\mathrm{P} + \mathrm{P})$. Their approach and our approach have several differences: two approaches take different input formats; their algorithm becomes significantly slower for tall slender trees whereas ours does not; the complexity of the Rake-Shunt contraction algorithm is $\mathrm{O}(\mathrm{N}/\mathrm{P} + \log \mathrm{P})$; thus, slightly better.

6. Conclusion and Future Work

We showed that efficient parallel tree contraction of binary trees is possible without $\mathrm{Compress}_{\mathrm{L}}^{\mathrm{R}}$, and such parallel tree contraction is useful to simplify discussions about parallel rose tree manipulations. This result can be a missing piece for generalizing parallel tree contraction for binary trees to rose trees.

We focused on theoretical analysis of parallel tree contraction and briefly discussed the practicality of $\mathrm{Compress}_{\mathrm{L}}^{\mathrm{R}}$-free parallel tree contraction. Further investigation in this direction is for future work. Our experiments were carried on a shared-memory machine; thus, experiments on different settings including distributed memory environments as well as comparison with different implementations including that by Kakehi et al. [6] are also left for the future.

Acknowledgments The authors are grateful to the reviewer for his/her comments, which were useful for improving the paper. The first author was supported by JSPS Grant-in-Aid for Young Scientists (B) 24700019.

References

[1] Abrahamson, K.R., Dadoun, N., Kirkpatrick, D.G. and Przytycka, T.M.: A Simple Parallel Tree Contraction Algorithm, *J. Algorithms*, Vol.10, No.2, pp.287–302 (1989).
[2] Brent, R.P.: The Parallel Evaluation of General Arithmetic Expressions, *J. ACM*, Vol.21, No.2, pp.201–206 (1974).
[3] Cole, R. and Vishkin, U.: The Accelerated Centroid Decomposition Technique for Optimal Parallel Tree Evaluation in Logarithmic Time, *Algorithmica*, Vol.3, pp.329–346 (1988).
[4] Diks, K. and Hagerup, T.: More General Parallel Tree Contraction: Register Allocation and Broadcasting in a Tree, *Theor. Comput. Sci.*, Vol.203, No.1, pp.3–29 (1998).
[5] Gazit, H., Miller, G.L. and Teng, S.-H.: Optimal tree contraction in the EREW model, *Proc. Princeton Workshop on Algorithms, Architectures, and Technology Issues for Models of Concurrent Computation*, pp.139–156, Plenum Press (1987).
[6] Kakehi, K., Matsuzaki, K. and Emoto, K.: Efficient Parallel Tree Reductions on Distributed Memory Environments, *Computational Science - ICCS 2007, 7th International Conference, Beijing, China, May 27-30, 2007, Proceedings, Part II*, Lecture Notes in Computer Science, Vol.4488, pp.601–608, Springer (2007).
[7] Kawamura, T. and Matsuzaki, K.: Dividing Huge XML Trees Using the m-bridge Technique over One-to-one Corresponding Binary Trees, *IPSJ Trans. Programming*, Vol.7, No.3, pp.40–50 (2014).
[8] Kosaraju, S.R. and Delcher, A.L.: Optimal Parallel Evaluation of Tree-Structured Computations by Raking, *VLSI Algorithms and Architectures, 3rd Aegean Workshop on Computing, AWOC 88, Corfu, Greece, June 28 - July 1, 1988, Proceedings*, Lecture Notes in Computer Science, Vol.319, pp.101–110, Springer (1988).
[9] Matsuzaki, K.: Efficient Implementation of Tree Accumulations on Distributed-Memory Parallel Computers, *Computational Science — ICCS 2007, 7th International Conference, Beijing, China, May 27–30, 2007, Proceedings, Part II*, Lecture Notes in Computer Science, Vol.4488, pp.609–616, Springer (2007).
[10] Matsuzaki, K.: Parallel Programming with Tree Skeletons, PhD Thesis, Graduate School of Information Science and Technology, The University of Tokyo (2007).
[11] Matsuzaki, K., Hu, Z., Kakehi, K. and Takeichi, M.: Systematic Derivation of Tree Contraction Algorithms, *Parallel Processing Letters*, Vol.15, No.3, pp.321–336 (2005).
[12] Matsuzaki, K., Hu, Z. and Takeichi, M.: Parallel skeletons for manipulating general trees, *Parallel Comput.*, Vol.32, No.7-8, pp.590–603 (2006).
[13] Miller, G.L. and Reif, J.H.: Parallel Tree Contraction and Its Application, *26th Annual Symposium on Foundations of Computer Science, 21-23 October 1985, Portland, Oregon, USA*, pp.478–489, IEEE (1985).
[14] Miller, G.L. and Teng, S.-H.: Tree-Based Parallel Algorithm Design, *Algorithmica*, Vol.19, No.4, pp.369–389 (1997).
[15] Morihata, A. and Matsuzaki, K.: A Practical Tree Contraction Algorithm for Parallel Skeletons on Trees of Unbounded Degree, *Proc. International Conference on Computational Science, ICCS 2011, Nanyang Technological University, Singapore, 1-3 June 2011*, Procedia CS, Vol.4, pp.7–16, Elsevier (2011).
[16] Peyton Jones, S.(Ed.): *Haskell 98 Language and Libraries: The Revised Report*, Cambridge University Press, Cambridge, UK (2003).

[17] Reif, J.H.(Ed.): *Synthesis of Parallel Algorithms*, Morgan Kaufmann Publishers (1993).
[18] Skillicorn, D.B.: Parallel Implementation of Tree Skeletons, *J. Parallel Distributed Comput.*, Vol.39, No.2, pp.115–125 (1996).

Akimasa Morihata received his Ph.D. from The University of Tokyo in 2009. He got JSPS research fellowships for young scientists (PD) in 2009, became an assistant professor at Tohoku University in 2010, and a lecturer at The University of Tokyo in 2014. His research interest includes functional programming, parallel programming, and transformational program development. He is a member of JSSST.

Kiminori Matsuzaki is an Associate Professor of Kochi University of Technology in Japan. He received his B.E., M.S. and Ph.D. from The University of Tokyo in 2001, 2003 and 2007, respectively. He was an Assistant Professor (2005–2009) in The University of Tokyo, before joining Kochi University of Technology as an Associate Professor in 2009. His research interest is in parallel programming and algorithm derivation. He is a member of ACM, JSSST, IEEE.