Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 4 (2011) 7–16

Procedia Computer Science

International Conference on Computational Science, ICCS 2011

A Practical Tree Contraction Algorithm for Parallel Skeletons on Trees of Unbounded Degree

Akimasa Morihata[a], Kiminori Matsuzaki[b]

[a]*Tohoku University, 2-1-1, Katahira, Aoba-ku, Sendai-shi, Miyagi, Japan*
[b]*Kochi University of Technology, 185, Miyanokuchi, Tosayamada-cho, Kochi, Japan*

Abstract

Algorithmic skeletons are ready-made parallel computation patterns. Since each skeleton can be evaluated efficiently on parallel computation environments, we can develop efficient parallel programs only by specifying our computation by a combination of skeletons. Although effectiveness of algorithmic skeletons, especially those manipulating arrays and lists, is now well-recognized, those for trees of unbounded degree have not been firmly established. Most of the existing studies transform them to binary trees through preprocessing. But this approach is not practical. The transformation not only makes developments of parallel programs difficult but also affects the performance of the parallel programs developed.

In this paper, we propose a parallel tree contraction algorithm named *Rake-Shunt contraction algorithm*. It is generalization of the Shunt contraction algorithm, which has been used as the algorithmic basis of the binary-tree skeletons, and inherits several good properties such as scalability with respect to the number of processors and simplicity of the implementation. Moreover, it can deal with arbitrary trees and requires no modification of their shapes. Our preliminary experiments show that our algorithm leads to an implementation of the tree reduction, one of the most important tree skeletons, that is easier to use and more efficient than the previous method based on the transformation to binary trees.

Keywords: Algorithmic Skeletons, Trees, Parallel Tree Contraction

1. Introduction

Algorithmic skeletons [1] (also known as *parallel skeletons*) are ready-made parallel computation patterns. Since each skeleton can be evaluated efficiently on parallel computation environments, we can develop efficient parallel programs only by specifying our computation by a combination of skeletons. Now effectiveness of algorithmic skeletons, especially those manipulating arrays and lists, is well-recognized. Familiar frameworks such as Google's MapReduce [2] and Intel® Threading Building Blocks [3] provide parallel skeletons.

Algorithmic skeletons for trees have been also considered. Skillicorn et al. [4, 5] proposed tree skeletons, and Matsuzaki et al. implemented a set of tree skeletons in parallel skeleton library SkeTo [6]. Yet, most researches

Selection and/or peer-review under responsibility of Prof. Mitsuhisa Sato and Prof. Satoshi Matsuoka
doi:10.1016/j.procs.2011.04.002

consider binary trees, and only a few discussed trees of unbounded degree. This is not satisfactory because most of the applications of trees, such as tree maps (i.e., dictionaries) and syntax trees, could be nonbinary. Above all, XML documents, which would be the most important trees in parallel computation, can be arbitrary shapes.

One of the most difficult parts in developing tree skeletons is to provide their efficient parallel evaluation algorithms. Concerning parallel evaluation, tree skeletons rely on *parallel tree contraction algorithms* [7, 8]. There is a simple and efficient parallel tree contraction algorithm for binary trees, called the Shunt contraction algorithm [8, 9, 10], but there appears to be no parallel tree contraction algorithms that are practical and can deal with arbitrary trees.

The standard method of dealing with nonbinary trees in parallel tree contraction is to transform the trees to binary trees [10, 11, 12, 13, 14] (sometimes called "binarization"), and indeed, Matsuzaki et al. [15] proposed an implementation of generic tree skeletons based on binarization. Their skeletons first transform the input tree to full binary trees and then perform parallel computations by using binary-tree skeletons. Apparently such an approach has several shortcomings. First, binarization is costly, especially when we would like to combine the skeletons with other computations. Second, binarization makes trees larger and thus requires more computational resources. Moreover, use of binary-tree skeletons on nonbinary trees is inconvenient. We should provide operators that simulate computations of generic trees on binary trees, and necessity of such unnatural operators does not only make development of parallel programs hard but also affects performance.

In order to overcome these problems, we propose a parallel tree contraction algorithm named *Rake-Shunt contraction algorithm*. It is a generalization of the Shunt contraction algorithm, and inherits several good properties such as scalability with respect to the number of processors and simplicity of the implementation. Moreover, it has a distinctive feature that it can deal with arbitrary trees but requires no modification of their shapes.

Although the algorithm brings neither new computational bounds nor new parallelizable problems, it would be useful for providing practical implementations based on parallel tree contraction, including those for generic tree skeletons. Indeed, our preliminary experiments show that our algorithm leads to an implementation of the tree reduction, one of the most important tree skeletons, that is easier to use and more efficient than the previous binarization-based method.

2. Preliminary

2.1. Trees and Basic Notations

As the input, we consider rooted trees each of whose node could have unbounded number of children. Nodes that have no children are called leaves, and internal nodes otherwise. We call nodes that have no sibling solitary.

In order to provide efficient implementation of our algorithm, we assume that the input tree is structured as follows. All of its nodes are stored in an array. Each node has three pointers: one to its first child, another to its next sibling, and the other to its "previous" node: each first child has a pointer to its parent, and other nodes have those to their previous sibling. In addition, each node can store a value. This is a standard implementation of nonbinary trees by using adjacency lists.

We adopt EREW-PRAM (Exclusive-Read Exclusive-Write Parallel Random Access Machines) as the model of our parallel computation. We use P and N to denote the number of processors and the size of the input tree (i.e., the number of nodes in the tree), respectively.

We use the lambda notations for representing functions. For example, $(\lambda x\, y.\, x + y)$ is the function that calculates the sum of its two operands.

2.2. Parallel Tree Contraction

Tree contraction [7, 8] is a problem to collapse a tree by successive applications of primitive contraction operations. An efficient parallel tree contraction algorithm leads to several efficient parallel algorithms on trees because it provides efficient and conflict-free scheduling of gathering values in the tree.

Figure 1 shows the three primitive contraction operations. Rake is an operation applied to a leaf, and eliminates it. Compress is applied to a solitary node and eliminates it. Shunt is applicable for a node that has exactly one sibling, and performs consecutive applications of Rake and Compress. The original parallel tree contraction algorithm, developed by Miller and Reif [7], was based on Rake and Compress. Later some researchers [9, 10] pointed out that Shunt leads to a simple and efficient parallel tree contraction on binary trees.

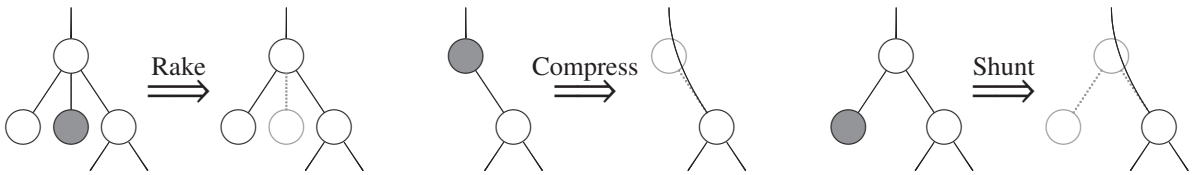

Figure 1: Primitive contraction operations applied to the gray-colored nodes: Rake (left), Compress (middle), and Shunt (right).

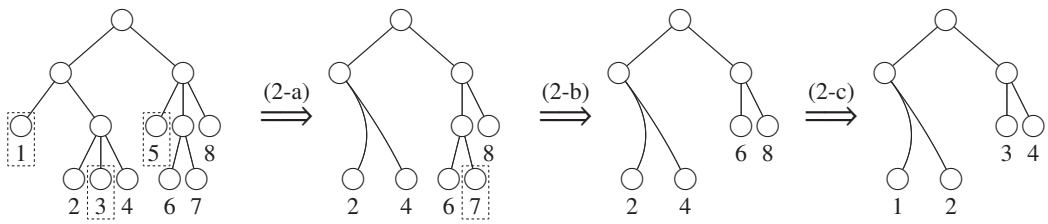

Figure 2: A behavior of Procedure 1: It describes an iteration of Steps (2a)–(2c): the dashed box specifies the leaves to which contraction operations are applied to at the step.

3. Rake-Shunt Contraction Algorithm

Here we propose the Rake-Shunt contraction algorithm. It consists of successive applications of Rake and Shunt operations. We call a Shunt operation Shunt-L if it is applied to the left leaf, and Shunt-R otherwise. Similarly, we call a Rake operation Rake-L/Rake-R if it merges the value of a leaf to its right/left sibling, respectively. It is worth noting that Shunt-L/Shunt-R are successive applications of Rake-L/Rake-R and Compress.

We assume that the input contains no solitary node; otherwise, we can make it binary by inserting a "dummy" leaf as its sibling. Dummy leaves are used only for scheduling and do not affect any computation. This assumption is for simplicity of explanation. Later we will show that it is not necessary to introducing dummy leaves.

The following is the Rake-Shunt contraction algorithm. This is a generalization of the Shunt contraction algorithm in the sense that the two algorithms are identical when the input is a full binary tree. We depict its behavior in Figure 2.

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

1. Number all leaves from left to right starting from 1.
2. Iterate Steps (2a)–(2c) until the tree becomes a leaf.
   (a) For all odd-numbered leaves having right siblings, apply Shunt-L if possible or Rake-L otherwise.
   (b) For all odd-numbered leaves, apply Shunt-R if possible or Rake-R otherwise.
   (c) Halve all the numbers.

In parallel tree contraction, we should avoid conflicts of contraction operations. For example, if we simultaneously apply Compress operations to both a node and its child, the tree structure will be broken. To avoid conflicts, the Shunt contraction algorithm [8, 9, 10] numbers leaves and apply Shunt operations separately to left and right leaves. We followed this approach.

In addition, we should be careful about implementation of Shunt operations. Figure 3 shows a critical case. The input tree is the one in the middle, and then, we apply a Shunt-L and a Rake-L operations to leaves 1 and 3, respectively. If the Shunt-L operation removes the parent of 1, as in Figure 1, then the left tree is resulted in. Since our tree is implemented by a doubly-linked adjacency lists, we should set the previous node of leaf 4 to be leaf 2; however, there is no way of knowing that the parent of leaf 2 is eliminated. In order to avoid this problem, we adopt another implementation, which results in the right tree in Figure 3: we eliminate the sibling instead of the parent. This implementation does not raise any dangerous pointer manipulations, even if the sibling has children.

Lemma 1. *Procedure 1 does not cause any conflicting applications of contraction operations; namely, no contraction operations will read/write values/addresses from nodes that are to be eliminated in the same step.*

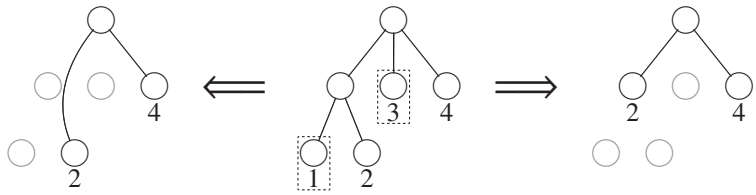

Figure 3: Critical applications of primitive contraction operations: the left describe the case where Shunt-L removes the parent, and the right describes the case where Shunt-L removes the sibling.

Proof. First of all, note that throughout the procedure, leaves are numbered from left to right starting from 1.

Here we only consider Step (2a). The case of Step (2b) is similar.

It is easy to see that two Rake-L operations do not conflict.

The reason that two Shunt-L operations do not conflict is the same as the case of binary trees [9, 10]. Assume that two Shunt-L operations to leaves n_1 and n_2 conflict. Without loss of generality, we can assume that the Shunt-L operation for n_2 tries to access a node that is to be eliminated by the Shunt-L operation for n_1. The only possibility is that the parent of n_2 is the sibling of n_1. However, it cannot occur because both the number of n_1 and n_2 should be odd, whereas no other leaf exists between them.

Lastly, we assume that a Shunt-L operation to leaf n_1 conflicts with a Rake-L operation to n_2. There are two cases. (i) the Shunt-L operation tries to access a node that is to be eliminated by the Rake-L. Rake-L cannot eliminate the parent of n_1 because the Rake-L operation eliminates n_2 that is a leaf. Therefore, the only possibility is that n_2 is the sibling of n_1. Apparently, it contradicts the fact that both n_1 and n_2 are odd valued. (ii) the Rake-L operation tries to access a node that is to be eliminated by the Shunt-L. Because of the requirement of applying Shunt-L, the only possibility is applying Rake-L to the first child of the right sibling of n_1. This cannot occur because of exactly the same reason as the case of two Shunt-L operations.

Note that it does not raise any conflict to determine which of Shunt-L and Rake-L we should apply, because it only access the nodes that possibly concern the contraction operations. □

Theorem 2. *Procedure 1 finishes in time* $\mathrm{O}(\mathrm{N}/\mathrm{P} + \log \mathrm{N})$.

Proof. Correctness of Procedure 1 follows from Lemma 1.

Step (1) finishes in the required time by using the list-ranking algorithm in combination with the Euler-tour technique [8]. It is sufficient to iterate Step (2) $\lceil \log_2 \mathrm{N} \rceil$ times, because each iteration of Steps (2a)–(2c) eliminates all odd-numbered leaves and the procedure terminates when all leaves but one are eliminated. Based on this observation, the computational complexity follows from Brent's scheduling theorem [16]. □

Avoidance of Introducing Dummy Leaves

The Rake-Shunt contraction algorithm does not require any modification on the shape of input trees as preprocessing, because it is actually unnecessary to introduce dummy leaves. Dummy leaves are only used for scheduling applications of Shunt operations. We can simulate this scheduling by appropriately numbering solitary nodes, instead of dummy leaves, and apply Compress operations to parents of odd-numbered solitary nodes.

To be concrete, we refine the algorithm as follows. In Step (1), we number all leaves and solitary nodes. The numbering should be in-order, and each solitary leaf should have two numbers. In Step (2a), we deal with each node that has an odd-number, say k, and satisfies either (i) it has a right sibling, or (ii) it is solitary leaf and numbered by both k and $k + 1$. We deal with the other odd-numbered nodes in Step (2b).

4. Implementing Tree Reductions by Rake-Shunt Contraction Algorithm

4.1. Tree Reductions

In order to demonstrate effectiveness of the Rake-Shunt contraction algorithm, here we propose an implementation of *tree reductions* [5, 14, 15], which is one of the most important tree skeletons.

A tree reduction on unbounded-degree trees gathers values by using two operators, say $\otimes$ and $\oplus$: It merges the value of siblings by $\oplus$, and calculates the contribution of children by $\otimes$. $\mathsf{reduce}_{\oplus,\otimes}$ denotes the tree reduction by $\oplus$ and $\otimes$ as follows.

$$\begin{array}{lcl} \mathsf{reduce}_{\oplus,\otimes}(\mathit{Node}(v,[\,])) & = & v \\ \mathsf{reduce}_{\oplus,\otimes}(\mathit{Node}(v,[t_1,\ldots,t_k])) & = & v \otimes (\mathsf{reduce}_{\oplus,\otimes}(t_1) \oplus \cdots \oplus \mathsf{reduce}_{\oplus,\otimes}(t_k)) \end{array}$$

Here $\mathit{Node}(v,[t_1,\ldots,t_k])$ denotes a tree whose root retains value v and children $t_1,\ldots,t_k$, and hence, $\mathit{Node}(v,[\,])$ is a leaf that consists of value v. We assume $\oplus$ to be associative.

4.2. Tree Reduction by Merging Partial Computations

The Rake-Shunt contraction algorithm gathers values in the tree according to the primitive contraction operations; therefore, in order to implement a tree reduction, we should decompose its calculation to computations that correspond to the primitive contraction operations, namely Rake and Compress. The computations merge the value of the node that is to be eliminated into that of either its sibling (Rake-L/Rake-R) or its children (Compress), and put it to the node left.

It is easy to develop the one that corresponds to Rake-L/Rake-R operations merging two leaves: calculate $v_1 \oplus v_2$ where v_1 and v_2 are the values in the leaves merged. The difficult case is that the sibling is an internal node. We cannot calculate the value of the sibling until the subtree rooted by the sibling is completely reduced.

The key is to consider merging partial computations. Note that internal nodes represent not values but partial computations. More concretely, an internal node having a value v corresponds to a function

$$(\lambda x\, l\, r.\, l \oplus (v \otimes x) \oplus r),$$

where x, l, and r respectively express the contribution of its children, the left siblings, and the right siblings. Then, if a Rake-L operation is applied to its left sibling that has v_l, the partial computation should be updated to the following.

$$(\lambda x\, l\, r.\, l \oplus v_l \oplus (v \otimes x) \oplus r)$$

We do not want to retain very large partial computations, and thus, we would like to shrink them as much as possible. For example, if another left sibling consisting of v'_l joins to the node above, we have

$$(\lambda x\, l\, r.\, l \oplus v'_l \oplus v_l \oplus (v \otimes x) \oplus r),$$

which can be shrunk to

$$(\lambda x\, l\, r.\, l \oplus v^*_l \oplus (v \otimes x) \oplus r)$$

where $v^*_l = v'_l \oplus v_l$. Similarly, we may try to shrink the partial computations in the case of Compress. If we are able to perform these shrinking, the tree reduction can be efficiently implemented by using the Rake-Shunt contraction algorithm.

Theorem 3. $\mathsf{reduce}_{\oplus,\otimes}$ *can be calculated in* $\mathrm{O}(\mathrm{N}/\mathrm{P} + \log \mathrm{N})$ *time by Procedure 1, provided that* $\oplus$ *and* $\otimes$ *are constant-time operations,* $\oplus$ *is associative, and there exists a constant-time function* $\rho_{\oplus,\otimes}$ *such that* $\rho_{\oplus,\otimes}(a,b,c;a',b',c') = (a^*,b^*,c^*)$ *implies that* $(\lambda x.\, a \oplus (b \otimes x) \oplus c) \circ (\lambda x.\, a' \oplus (b' \otimes x) \oplus c') = (\lambda x.\, a^* \oplus (b^* \otimes x) \oplus c^*)$.

Proof. In what follows, we regard the associated value as the identifier of the node.

Initially, we associate each internal node v with function $(\lambda x.\, \iota_\oplus \oplus (v \otimes x) \oplus \iota_\oplus)$, where $\iota_\oplus$ is the unit of $\oplus$. Note that even if $\oplus$ does not have the unit, we can introduce it by providing an additional flag that denotes whether the value is the unit.

Next, we apply the Rake-Shunt contraction algorithm. In each primitive contraction operation, we update the associated values as follows. We omit the cases of Rake-R because they are very similar to those of Rake-L.

- When Rake-L is applied to leaf v and its left sibling is leaf v', associate the leaf left with $v \oplus v'$.
- When Rake-L is applied to leaf v and its left sibling is internal node $(\lambda x.\, a \oplus (b \otimes x) \oplus c)$, associate the internal node left with $(\lambda x.\, a' \oplus (b \otimes x) \oplus c)$, where $a' = v \oplus a$.

- When Compress is applied to solitary node $(\lambda x.\, a \oplus (b \otimes x) \oplus c)$ and its child is leaf v, associate the leaf left with v' where $v' = a \oplus (b \otimes v) \oplus c$.
- When Compress is applied to solitary node $(\lambda x.\, a \oplus (b \otimes x) \oplus c)$ and its child is internal node $(\lambda x.\, a' \oplus (b' \otimes x) \oplus c')$, associate the internal node left with $(\lambda x.\, a^* \oplus (b^* \otimes x) \oplus c^*)$ where $(a^*, b^*, c^*) = \rho_{\oplus,\otimes}(a, b, c; a', b', c')$.

Correctness of the computation is evident, and the computational complexity follows from Theorem 2. □

Theorem 3 provides an implementation of $\mathsf{reduce}_{\oplus,\otimes}$. This implementation brings parallel computation if we succeed in developing $\rho_{\oplus,\otimes}$.

4.3. Example 1: Maximum Path Weight

Now let us demonstrate use of Theorem 3. The first example is the maximum path weight problem. We would like to find the maximum-weighted path from the root to a leaf, where the weight of a path is the sum of values on the path. This is a kind of problem of finding the critical path.

It is easy to specify the problem as a tree reduction: $\mathsf{reduce}_{\uparrow,+}$, where $\uparrow$ is the binary maximum operator. Therefore, according to Theorem 3, we consider partial computations of the form, $(\lambda x.\, a \uparrow (b+x) \uparrow c)$. Note that $\uparrow$ is commutative; thus, for this problem, simpler $(\lambda x.\, a \uparrow (b + x))$ should be sufficient.

Now let us confirm the premise of Theorem 3.

$$
\begin{array}{lcl}
(\lambda x.\, a \uparrow (b + x)) \circ (\lambda x.\, a' \uparrow (b' + x)) & = & \{ \text{ function composition } \} \\
 & & (\lambda x.\, a \uparrow (b + (a' \uparrow (b' + x)))) \\
 & = & \{\ u + (v \uparrow w) = ((u + v) \uparrow (u + w))\ \} \\
 & & (\lambda x.\, a \uparrow (b + a') \uparrow (b + b' + x)) \\
 & = & \{ \text{ simplification } \} \\
 & & (\lambda x.\, a^* \uparrow (b^* + x)) \quad \textbf{where } a^* = a \uparrow (b + a') \text{ and } b^* = b + b'
\end{array}
$$

The reasoning gives the definition of $\rho_{\uparrow,+}$, which takes only four arguments in this case: $\rho_{\uparrow,+}(a, b; a', b') = (a \uparrow (b + a'), b + b')$. Therefore, by associating two values for each internal node, we can compute the maximum path weight efficiently in parallel.

It is worth noting that in the reasoning above, we only used algebraic properties of $\uparrow$ and $+$, namely, the associativity and commutativity of $\uparrow$, the associativity of $+$, and the distributivity of $+$ over $\uparrow$. Note that they are exactly the properties of semirings. In other words, the same evaluation schema is applicable to any tree reduction that is specified by a semiring.

4.4. Maximum Subtree Sum

The other example is the maximum subtree sum problem. We would like to calculate the maximum weight of a subtree, and the weight is the sum of values in the subtree. This problem is a kind of query of extracting the most meaningful part.

It is not difficult to express the maximum subtree sum by a tree reduction. Since we also need to calculate weights of subtrees, the reduction yields two values: $\mathsf{reduce}_{\oplus,\otimes}$ where $(m_1, s_1) \oplus (m_2, s_2) = (m_1 \uparrow m_2, s_1 + s_2)$ and $v \otimes (m, s) = ((v + s) \uparrow m, v + s)$. The first component retains the maximum subtree sum, whereas the second is the weight of the subtree. We update the maximum subtree sum when the weight of the subtree is larger than the current candidate of the maximum.

It is easy to see associativity of $\oplus$; moreover, $\oplus$ is commutative. Thus, it is sufficient to consider $(\lambda x.\, a \oplus (b \otimes x))$.

We derive $\rho_{\oplus,\otimes}$ as follows.

$$
\begin{aligned}
&(\lambda x.\,(m_a, s_a) \oplus (b \otimes x)) \circ (\lambda x.\,(m_{a'}, s_{a'}) \oplus (b' \otimes x)) \\
&\quad = \quad \{ \text{ function composition } \} \\
&\qquad (\lambda x.\,(m_a, s_a) \oplus (b \otimes ((m_{a'}, s_{a'}) \oplus (b' \otimes x)))) \\
&\quad = \quad \{ \text{ let } (m_x, s_x) = x \text{; definitions of } \oplus \text{ and } \otimes \} \\
&\qquad (\lambda(m_x, s_x).\,(m_a \uparrow (b + s_{a'} + b' + s_x) \uparrow m_{a'} \uparrow (b' + s_x) \uparrow m_x, s_a + b + s_{a'} + b' + s_x)) \\
&\quad = \quad \{ \text{ simplification by using algebraic properties of } \uparrow \text{ and } + \} \\
&\qquad (\lambda(m_x, s_x).\,((m_a \uparrow m_{a'}) \uparrow (((b + s_{a'} + b') \uparrow b') + s_x) \uparrow m_x, (s_a + b + s_{a'} + b') + s_x)) \\
&\quad = \quad \{ \text{ definitions of } \oplus \text{ and } \otimes \} \\
&\qquad (\lambda(m_x, s_x).\,a^* \oplus (b^* \otimes (m_x, s_x))) \\
&\qquad \textbf{where } a^* = (m_a \uparrow m_{a'}, s_a + b + s_{a'} + b' - b^*) \text{ and } b^* = (b + s_{a'} + b') \uparrow b'
\end{aligned}
$$

We succeeded in developing $\rho_{\oplus,\otimes}$; therefore, from Theorem 3, we can calculate the maximum subtree sum in parallel by associating each internal node with three numbers.

4.5. Discussions

Theorem 3 shows that $\mathsf{reduce}_{\oplus,\otimes}$ can be implemented by using the Rake-Shunt contraction, if we provide a function $\rho_{\oplus,\otimes}$. Note that known results [8, 12, 13] guarantee the same computational complexity from the premise. The important point is that our parallel programs do not require any modification of the shape of the input tree.

The previous implementation [15] also requires a similar function, and $\rho_{\oplus,\otimes}$ is easier to develop. [15] uses binarization, which requires to develop operations that *simulate* the computations on binary trees. Our implementation does not have such difficulty.

This simplicity is also beneficial from the perspective of efficiency of developed programs. Our implementation often requires fewer primitive values and fewer units for representing partial computations. For instance, for computing the maximum path weight, the previous method requires three numbers and two units, namely those for $+$ and $\uparrow$. For the maximum subtree sum, four numbers and three units, that of $\oplus$ and the left and the right units of $\otimes$, are necessary. It is likely that simulating the computation on binary trees requires more information for retaining the partial computations. Indeed, for all problems considered in [14, 15, 21], our method requires fewer primitive values, or at least not more, than for the previous method.

It is also worth noting that the premise of Theorem 3 is very similar to those considered in the literature [4, 8, 10, 15, 17, 18, 19, 20], and we can mechanically develop $\rho_{\oplus,\otimes}$ based on the existing parallelization methods [17, 20].

5. Experiments

We carried out an experiment so as to evaluate the efficiency of our implementation of the tree reduction. Its environment consists of dual quad-core Xeon X5550 2.66-GHz CPUs, 12-GB memory, 64-bits Linux 2.6.32 (Ubuntu 10.04), and ICC 11.1 with the -O2 option.

We considered two problems in the previous section, namely the maximum path weight problem and the maximum subtree sum problem. For each problem, we developed four programs based on C++ with OpenMP: SEQ is a sequential recursion-free program; RS is based on the Rake-Shunt contraction algorithm; [15]-ND implements the previously-proposed method [15], but it does not introduce any dummy leaves; [15]-WD implements the previously-proposed method [15] by using dummy leaves. Note that the method [15] relies on the implementation of binary tree skeletons. We adopt the Shunt contraction algorithm as originally suggested [5][1].

We randomly generated two input trees. Each tree consists of 2^{25} nodes, and each node contains a 64-bits integer from -2^{19} to $2^{19} - 1$. TALL is a slender tall tree whose height is $2^{20} + 1$. FLAT is a flat short tree whose height is 9.

Tables 1–4 summarize the results. The computational times do not include those for inputting and preprocessing the tree, such as those for modifying the tree shape and numbering the leaves.

First, RS became slightly faster than SEQ when we used 8 cores. This implies that our approach is promising.

[1] [15] considers distributed environments and thus it cannot directly be compared to our method.

Table 1: Running times for computing the maximum path weight of FLAT (unit: second).

	P = 1	P = 2	P = 3	P = 4	P = 6	P = 8
SEQ	0.53					
RS	1.06	0.74	0.66	0.54	0.51	0.49
[15]-ND	1.58	0.73	0.67	0.53	0.50	0.49
[15]-WD	2.86	1.29	1.12	0.91	0.82	0.70

Table 2: Running times for computing the maximum path weight of TALL (unit: second).

	P = 1	P = 2	P = 3	P = 4	P = 6	P = 8
SEQ	0.63					
RS	2.05	1.05	0.90	0.75	0.64	0.62
[15]-ND	2.61	1.29	1.12	0.90	0.74	0.68
[15]-WD	4.15	2.00	1.71	1.38	1.19	1.09

Table 3: Running times for computing the maximum subtree sum of FLAT (unit: second).

	P = 1	P = 2	P = 3	P = 4	P = 6	P = 8
SEQ	0.81					
RS	1.72	0.83	0.73	0.60	0.56	0.55
[15]-ND	1.83	0.84	0.64	0.60	0.56	0.54
[15]-WD	3.30	1.49	1.26	1.03	0.92	0.86

Table 4: Running times for computing the maximum subtree sum of TALL (unit: second).

	P = 1	P = 2	P = 3	P = 4	P = 6	P = 8
SEQ	0.79					
RS	2.41	1.18	1.01	0.87	0.71	0.68
[15]-ND	2.96	1.47	1.27	1.00	0.86	0.76
[15]-WD	4.70	2.26	1.96	1.54	1.34	1.22

Nevertheless, the speedup does not match the theoretical complexity. One of the reasons is the imbalance of tasks. Even though every processor deals with nearly the same number of leaves in Steps (2a)–(2c), those considered in Step (2a) or (2b) could vary: a processor may come across only the rightmost leaves, whereas another may have no such. In order to resolve this imbalance, the current implementation divides leaves into $16P$ chunks, and dynamically assign each chunk to a processor that has finished its task.

Even if we take the task imbalance into account, the speedups do not seem very nice. We suspect that memory operations form a bottleneck. Parallel tree contraction algorithms irregularly access nodes, and the irregularity may reduce memory throughput. For providing more efficient implementations, a considerable approach is to combine the Rake-Shunt contraction algorithm with the *m*-bridge technique [8, 22]. The technique divides a tree into a set of segments, each of which can be collapsed by using the primitive contraction operations, and thus, reduces the number of memory operations. It will be also useful for providing an implementation on distributed-memory environments.

Next, let us compare the implementations. The results for two problems were very similar. [15]-WD was apparently slower, which showed that it was effective to avoid introducing dummy leaves. [15]-ND was as fast as RS on FLAT, but it was slower on TALL. Since the implementation of [15] requires more numbers for retaining partial computations, it is natural that [15]-ND is slower than RS. This overhead is not apparent on FLAT because the most costly operation, namely merging of two partial computations, rarely occurs. It is worth noting that RS uses less space than others because it uses fewer number of values. In summary, we can conclude that RS is the most preferable.

6. Related Works

There have been a lot of studies on parallel tree contraction algorithms, including their applications to trees that have unbounded number of children [7, 8, 10, 11, 12, 13, 14, 23, 24]. Yet, most of the results have been developed from algorithmic viewpoints, and only a few [22, 25] discussed practically efficient implementations. We have

developed practically efficient parallel tree contraction that can deal with arbitrary trees.

The original parallel tree contraction by Miller and Reif [7] is applicable for arbitrary trees, but it requires concurrent read and write. A standard method of adapting it to exclusive-read/write machines is to use its randomized variant. The randomized variant might be practical, while deterministic algorithms can guarantee worst-case complexity.

Another well-known method of dealing with arbitrary trees is to binarize them [10, 11, 13, 15]. Though this approach is effective in theory, it is less practical as our experiment shows. Abrahamson et al. [10] proposed a method of binarization, and claimed that the binarization is just a reinterpretation of pointers and thus introduces no cost. Yet, the binarization makes all the nodes to be leaves and thereby makes the tree virtually larger; moreover, it transforms the tree shape and therefore makes developments of parallel programs difficult.

Miller and Teng [12] proposed a tree-contraction-based method of evaluating expressions that may form non-binary operators. Their method does not perform binarization. Instead, in order to achieve conflict-free scheduling of primitive contraction operations, their method numbers leaves for every iteration of applying Shunt operations. Apparently, this numbering introduces additional costs.

7. Conclusion and Future Works

We have developed a new parallel tree contraction algorithm, the Rake-Shunt contraction algorithm. It is a variant of the Shunt contraction algorithm [8, 9, 10] but applicable to arbitrary trees without modifying the shape of the input. It leads to an implementation of tree reductions, which is easier to use and more efficient than the previous one.

Regrettably, there are computations that Theorem 3 cannot cope with. For instance, Miller and Reif [23] and Diks et al. [13] showed that parallel tree contraction algorithms are potentially able to deal with those in which computations corresponding to Rake operations cannot be performed incrementally, i.e., we cannot start a computation for an internal node until almost all of its children are available. We would like to check whether our method is also extensible to them.

The absence of binarization might be effective when we consider combination of other parallel programming technique. One possibility is a combination with flattening technique [26, 27], in which a tree is represented by a nested array. Another possibility is the method by Kakehi et al. [24], which is a parallel tree contraction without parsing, namely calculating a value directly from a string that consists of brackets encoding the input tree. We believe that such combinations could bring interesting results.

We have only considered tree reductions. Tree skeletons contain more involved computations, namely tree accumulations [4]. A standard method of implementing tree reductions is to rewind the tree contraction procedure [4, 10, 12]. Indeed, we can implement tree accumulations by the Rake-Shunt contraction algorithm based on this strategy. This approach would provide a simpler implementation. The previous one [15] is rather complicated because it should be careful that the binarization does not affect the result. Our algorithm does not introduce such a trouble.

Acknowledgements. The authors are grateful to Kento Emoto, Shigeyuki Sato, and Zhenjiang Hu for their fruitful discussions on an early version of this work. The authors are also grateful for anonymous referees for their valuable comments.

References

[1] M. I. Cole, Algorithmic Skeletons: Structural Management of Parallel Computation, MIT Press, 1989.
[2] J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, in: 6th Symposium on Operating System Design and Implementation, 2004, pp. 137–150.
[3] J. Reinders, Intel threading building blocks, O'Reilly & Associates, Inc., 2007.
[4] J. Gibbons, W. Cai, D. B. Skillicorn, Efficient parallel algorithms for tree accumulations, Science of Computer Programming 23 (1) (1994) 1–18.
[5] D. B. Skillicorn, Parallel implementation of tree skeletons, Journal of Parallel and Distributied Computing 39 (2) (1996) 115–125.
[6] K. Matsuzaki, H. Iwasaki, K. Emoto, Z. Hu, A library of constructive skeletons for sequential style of parallel programming, in: Proceedings of the 1st International Conference on Scalable Information Systems, Infoscale 2006, Vol. 152 of ACM International Conference Proceeding Series, 2006, p. 13.
[7] G. L. Miller, J. H. Reif, Parallel tree contraction and its application, in: 26th Annual Symposium on Foundations of Computer Science, 1985, pp. 478–489.

[8] J. H. Reif (Ed.), Synthesis of Parallel Algorithms, Morgan Kaufmann Publishers, 1993.
[9] S. R. Kosaraju, A. L. Delcher, Optimal parallel evaluation of tree-structured computations by raking, in: VLSI Algorithms and Architectures, 3rd Aegean Workshop on Computing, AWOC 88, Proceedings, Vol. 319 of Lecture Notes in Computer Science, 1988, pp. 101–110.
[10] K. R. Abrahamson, N. Dadoun, D. G. Kirkpatrick, T. M. Przytycka, A simple parallel tree contraction algorithm, Journal of Algorithms 10 (2) (1989) 287–302.
[11] R. Cole, U. Vishkin, The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time, Algorithmica 3 (1988) 329–346.
[12] G. L. Miller, S.-H. Teng, Tree-Based Parallel Algorithm Design, Algorithmica 19 (4) (1997) 369–389.
[13] K. Diks, T. Hagerup, More general parallel tree contraction: Register allocation and broadcasting in a tree, Theoretical Computer Science 203 (1) (1998) 3–29.
[14] K. Matsuzaki, Z. Hu, K. Kakehi, M. Takeichi, Systematic derivation of tree contraction algorithms, Parallel Processing Letters 15 (3) (2005) 321–336.
[15] K. Matsuzaki, Z. Hu, M. Takeichi, Parallel skeletons for manipulating general trees, Parallel Computing 32 (7-8) (2006) 590–603.
[16] R. P. Brent, The parallel evaluation of general arithmetic expressions, Journal of the ACM 21 (2) (1974) 201–206.
[17] A. L. Fisher, A. M. Ghuloum, Parallelizing complex scans and reductions, in: Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation, 1994, pp. 135–146.
[18] W.-N. Chin, A. Takano, Z. Hu, Parallelization via context preservation, in: Proceedings of the 1998 International Conference on Computer Languages, ICCL '98, 1998, pp. 153–162.
[19] K. Matsuzaki, Z. Hu, M. Takeichi, Parallelization with tree skeletons, in: Euro-Par 2003, Parallel Processing, 9th International Euro-Par Conference, Proceedings, Vol. 2790 of Lecture Notes in Computer Science, 2003, pp. 789–798.
[20] A. Morihata, K. Matsuzaki, Automatic parallelization of recursive functions using quantifier elimination, in: Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Proceedings, Vol. 6009 of Lecture Notes in Computer Science, 2010, pp. 321–336.
[21] K. Matsuzaki, Parallel programming with tree skeletons, Ph.D. thesis, Graduate School of Information Science and Technology, The University of Tokyo (2007).
[22] K. Matsuzaki, Efficient implementation of tree accumulations on distributed-memory parallel computers, in: Computational Science - ICCS 2007, 7th International Conference, Proceedings, Part II, Vol. 4488 of Lecture Notes in Computer Science, 2007, pp. 609–616.
[23] G. L. Miller, J. H. Reif, Parallel tree contraction, part 2: Further applications, SIAM Journal on Computing 20 (6) (1991) 1128–1147.
[24] K. Kakehi, K. Matsuzaki, K. Emoto, Efficient parallel tree reductions on distributed memory environments, in: Computational Science - ICCS 2007, 7th International Conference, Proceedings, Part II, Vol. 4488 of Lecture Notes in Computer Science, 2007, pp. 601–608.
[25] D. A. Bader, S. Sreshta, N. R. Weisse-Bernstein, Evaluating arithmetic expressions using tree contraction: A fast and scalable parallel implementation for symmetric multiprocessors (SMPs) (extended abstract), in: High Performance Computing - HiPC 2002, 9th International Conference, Proceedings, Vol. 2552 of Lecture Notes in Computer Science, 2002, pp. 63–78.
[26] G. E. Blelloch, J. C. Hardwick, J. Sipelstein, M. Zagha, S. Chatterjee, Implementation of a portable nested data-parallel language, Journal of Parallel and Distributed Computing 21 (1) (1994) 4–14.
[27] G. Keller, M. M. T. Chakravarty, Flattening trees, in: Euro-Par '98 Parallel Processing, 4th International Euro-Par Conference, Proceedings, Vol. 1470 of Lecture Notes in Computer Science, 1998, pp. 709–719.