DOI 10.12694/scpe.v18i1.1230
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 18, Number 1, pp. 1–15. http://www.scpe.org

EFFICIENT PARALLEL TREE REDUCTIONS ON DISTRIBUTED MEMORY ENVIRONMENTS *

KAZUHIKO KAKEHI† KIMINORI MATSUZAKI‡ AND KENTO EMOTO§

Abstract. A new approach for fast parallel reductions on trees over distributed memory environments is presented. The start point of our approach is to employ the serialized representation of trees. Along this data representation with high memory locality and ease of initial data representation, we developed an parallelized algorithm which shares the essence with the parallel algorithm for parentheses matching problems. Our algorithm not only is proven to be theoretically efficient, but also has a fast implementation in a BSP style.

Key words: Parallel tree reduction, parentheses matching, serialized representation, the tree contraction algorithm, Bulk Synchronous Parallelism.

AMS subject classifications. 68W10, 05C05

1. Introduction. Research and development of parallelized algorithms have been intensively done toward matrices or one dimensional arrays. Looking at recent trends in applications, another data structure has also been calling for efficient parallel treatments: the tree structures. Emergence of XML as a universal data format, which takes the form of a tree, has magnified the impact of parallel and distributed mechanisms toward trees in order to reduce computation time and mitigate limitation of memory.

Consider, as a simple and our running example, a computation *maxPath* to find the maximum of the values each of which is a sum of values in the nodes from the root to each leaf. When it is applied to the tree at the left of Fig. 2.2, the result should be 12 contributed by the path of values 3, −5, 6 and 8 from the root. Recalling the research on parallel treatments on trees, *the parallel tree contractions algorithm*, first proposed by Miller and Reif [34], have been known as one of the most fundamental techniques to realize parallel computation over trees efficiently. One attractive feature of parallel tree contractions is that no matter how imbalanced a tree is, the tree can be reduced in parallel to a single node by repeatedly contracting edges and merging adjacent nodes. This theoretical beauty, however, may not necessarily shine in practice, because of the following two source of problems.

First, parallel tree contractions, originally being developed under the assumption of shared memory environments, have been intensively studied in the context of the PRAM model of parallel computation. This assumption does not apply to the recent trends of popular PC clusters, a common and handy approach for distributed memory environments, where the treatment and cost of data arrangement among processors need taking into account. It should be noted that efficient tree contractions under PRAM model are realized by assigning contractions to different processors each time.

Second, the parallel tree contractions algorithm assumes that the tree structures are kept as linked structures. Such representations using links are not suitable for fast execution, since linked structures often do not fit in caching mechanism, and it is a big penalty on execution time under current processor architectures.

This paper gives a clear solution for parallel tree reductions with its start point to use *serialized forms of trees*. Their notable examples are the serialized (streamed) representations of XML or parenthesized numeric expressions which are obtained by tree traversals. The problems mentioned above are naturally resolved by this choice, since distribution of serialized data among processors and realization of routines running over them with high memory locality are much simpler than that of trees under linked structures.

Our algorithm over the serialized tree representations is developed along with the *parentheses matching*

*An earlier version of this paper appeared in PAPP2007, part of ICCS2007[23].

†Academic Co-Innovation Division, UTokyo Innovation Platform Co., Ltd., 3–40–10 Hongo, Bunkyo-ku, Tokyo 113-0033 Japan (k.kakehi@utokyo-ipc.co.jp)

‡School of Information, Kochi University of Technology, 185 Tosayamadacho-Miyanokuchi, Kami, Kochi 782–8502 Japan (matsuzaki.kiminori@kochi-tech.ac.jp)

§Faculty of Computer Science and Systems Engineering, Kyushu Institute of Technology, 680-4 Kawazu, Iizuka, Fukuoka 820-8502, Japan (emoto@ai.kyutech.ac.jp)

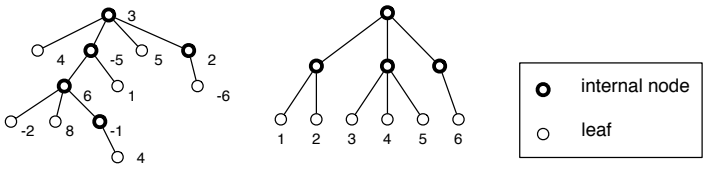


FIG. 2.1. *A rose tree (left), and the rose tree representation of* $[[1,2],[3,4,5],[6]]$ *(right)*

problems. As instances of serialized trees, parallelization of parentheses matching problems, which figures out correspondence between brackets, have plenty of work ([4, 36, 13, 25, 21] for example). Our algorithm, with good resemblance to the one under BSP [43], also has a BSP implementation with three supersteps.

The contributions of our work are briefly summarized as follows:

- *A New viewpoint at the parallel tree contractions algorithm—connection to parentheses matching*: We cast a different view on computing tree structures in parallel. The employed data representation, a serialized form of trees by tree traversals, has massive advantages in the current computational environments: namely XML as popular data representations, cache effects in current processor architectures, and ease in data distribution among popular PC clusters. Tree computations over its serialized representation has much in common with parentheses matching. It is common to see that this problem has connection with trees, like binary tree reconstruction or computation-tree generation, but to the best of our knowledge we are the first to apply the idea of parentheses matching toward parallel tree reductions, and to prove its success.
- *Theoretical and practical efficiency*: The previous work of parallel tree contractions on distributed memory environments, namely hypercube [32] or BSP [16], both of which require $O(N/P \log P)$ execution time. Contrasting to other work, our approach employs serialized tree representations. As a result, we realized an $O(N/P + P)$ algorithm.

This paper is organized as follows. After this Introduction Sect. 2 observes our tree representations and tree reductions. Section 3 explains an additional condition which enables efficient parallelization. We show that this condition is equivalent to that of the parallel tree contractions algorithm. Section 4 develops the parallelized algorithm. Our algorithm consists of three phases, where the first two perform computation along with parentheses matching, and the last reduces a tree of size less than twice of the number of processors. Section 5 supports our claims by some experiments. They demonstrate good scalability in computation. On the other hand, we also observe fluctuation of data transactions. We discuss the related work in Sect. 6. Finally we conclude this paper in Sect. 7 with mentioning future directions.

2. Preliminaries. This section defines the target of our parallelization, namely tree structures and their serialized form, and tree homomorphism.

2.1. Trees and their serialized representation. We treat trees with unbound degree (trees whose nodes can have an arbitrary number of subtrees), which is often referred as "*rose trees*" in functional programming communities.

DEFINITION 2.1. *A data structure RTree is called* rose tree *defined as follows.*

$$RTree\ \alpha = Node\ \alpha\ [Rtree\ \alpha]$$

Fig. 2.1 shows examples of rose trees The left tree is set to be used as our running example. Rose trees are general enough to represent nested lists; the right part of Fig. 2.1 is what $[[1,2],[3,4,5],[6]]$ is formatted into a rose tree, whose internal nodes do not have their value. Binary trees are also covered by limiting degrees of each node to be either zero or two.

As was explained in Introduction, our internal representation is to keep tree-structured data in a serialized manner like XML. For example, if we are to deal with a tree with just two nodes, parent `a` and its child `b`, the serialized equivalent is a list of four elements [`<a>`,`<b>`,`</b>`,`</a>`] . This is a combination of a preorder traversal (for producing the open tag `<a>` and then `<b>`) and a postorder traversal (for producing the close tag `</b>` and `</a>` afterwards). This representation has information enough to obtain back the original tree. In

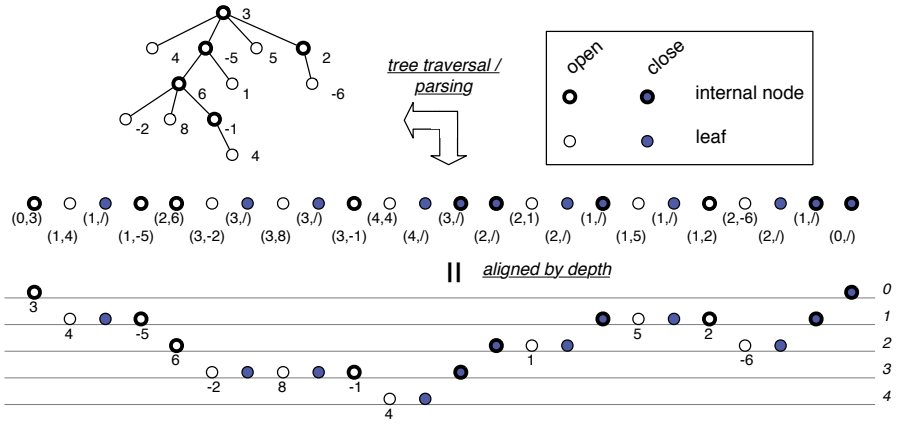


FIG. 2.2. *A rose tree (upper left), its serialized representation as a sequence of pairs (middle) and another representation according to the depth (lower)*

this paper we treat only *well-formed* serialized representation, which is guaranteed to be parsed into trees. To make such serialized representation easy to observe, we assume (1) to ignore information in the closing tag, and (2) to assign the information about the depth in the tree instead. We therefore assume to deal with a list of pairs [(0,a), (1,b), (1,/), (0,/)], where / denotes closing tags. The associated depth information can be easily obtained by prefix sum computation [5].

The sequence of the middle in Fig. 2.2 is our internal representation of the example tree. We later see the information of depth helps us to have insight for deriving parallelized algorithms. Following these figures, list elements describing the entrance (which corresponds to open tags) are called *open*, and ones describing the exit (close tags) are called *close*. These close elements without any value can be soon wiped away when we perform computation.

2.2. Tree homomorphism. Algorithms are often tightly connected with the data structure. A general recursive form naturally arises here, which we call *tree homomorphism* [42].

DEFINITION 2.2. *A function h defined as follows over rose trees is called* tree homomorphism.

$$\begin{aligned} h(\ \mathit{Node}\ a\ [t_1, \ldots, t_n]\ ) &= a \oplus (\ h(t_1) \otimes \cdots \otimes h(t_n)\ ) \\ h(\ \mathit{Leaf}\ a\ ) &= h'(a) \end{aligned}$$

We assume the above operator $\otimes$ is associative and has its unit $\iota_\otimes$.

This definition consists of a function h' for leaves, and two kinds of computation for internal nodes: $\otimes$ to reduce recursive results from subtrees into one, and $\oplus$ to apply the result to the internal node. For later convenience, we define $\ominus$ as $(a, b, c) \ominus e = a \oplus (b \otimes e \otimes c)$, and we call (a, b, c) appearing at the left of $\ominus$ a "*triple*". The computation *maxPath* mentioned in Introduction is also a tree homomorphism.

$$\begin{aligned} \mathit{maxPath}(\ \mathit{Node}\ a\ [t_1, \ldots, t_n]\ ) &= a + (\ \mathit{maxPath}(t_1) \uparrow \cdots \uparrow \mathit{maxPath}(t_n)\ ) \\ \mathit{maxPath}(\ \mathit{Leaf}\ a\ ) &= \mathit{id}(a) \\ &= a \end{aligned}$$

Here, *id* is the identify function, and $\uparrow$ returns the bigger of two numbers whose unit is $-\infty$. When it is applied to the tree in Fig. 2.2, the result should be $12 = 3 + (-5) + 6 + 8$. For other examples please see [27].

3. Parallelization Condition. When we develop parallel implementations we often utilize associativity of computations, in order to change the order of computation with guaranteeing the same result. The framework of tree homomorphism only requires the condition that the operator $\otimes$ is associative. This property of the *horizontal* directions is indeed necessary for parallelization, but not yet sufficient: we currently lack the effective measure toward computation of the *vertical* directions.

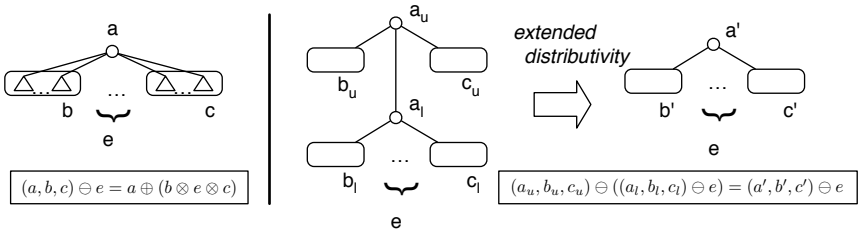


FIG. 3.1. *A* triple *(left) and extended distributivity (right)*

Before we start algorithm development, this section observes an additional property. What we use is known as *extended distributivity* [27]. This property is explained using the operator $\ominus$ as follows.

DEFINITION 3.1. *The operator* $\otimes$ *is said* extended distributive *over* $\oplus$ *when the following equation holds for any* a_u, b_u, c_u, a_l, b_l, c_l, *and* e*:*

$$\begin{aligned}(a_u,b_u,c_u) \ominus ((a_l,b_l,c_l) \ominus e) &= a_u \oplus (b_u \otimes (a_l \oplus (b_l \otimes e \otimes c_l)) \otimes c_u) \\ &= a' \oplus (b' \otimes e \otimes c') \\ &= (a',b',c') \ominus e\end{aligned}$$

with appropriate functions p_a, p_b *and* p_c *which calculate*

$$\begin{aligned}a' &= p_a(a_u,b_u,c_u,a_l,b_l,c_l)\ , \\ b' &= p_b(a_u,b_u,c_u,a_l,b_l,c_l)\ ,\ \textit{and} \\ c' &= p_c(a_u,b_u,c_u,a_l,b_l,c_l)\ .\end{aligned}$$

Efficient parallel tree reductions require these properties as well as $\otimes$ and $\oplus$ to be constant-time. Our running example satisfies them, implicitly with its characteristic functions $a' = a_u + a_l$, $b' = b_u - a_l \uparrow b_l$, $c' = c_l \uparrow c_u - a_l$, as the following calculation shows.

$$\begin{aligned}&(a_u,b_u,c_u) \ominus (\ (a_l,b_l,c_l) \ominus e\ ) \\ &= a_u + (b_u \uparrow (a_l + (b_l \uparrow e \uparrow c_l)) \uparrow c_u) \\ &= (a_u + a_l) + ((-a_l + b_u \uparrow b_l) \uparrow e \uparrow (c_l \uparrow -a_l + c_u)) \\ &= (\ (a_u + a_l),\ (-a_l + b_u \uparrow b_l),\ (c_l \uparrow -a_l + c_u)\ ) \ominus e\end{aligned}$$

The *triple* and extended distributivity from the viewpoint of their tree structures are depicted in Fig. 3.1. We show the property of extended distributivity is equivalent to the condition for realizing the parallel tree contractions algorithm with respect to rose trees. Parallel tree contractions require two operations *rake* and *compress* are efficiently computed. The operation *rake* is to contract an edge between a leaf node and its parent internal node; the other operation *compress* is to contract the last remaining edge of an internal node. Once these treatments are possible, tree computations are performed in parallel (see Fig. 3.2).

THEOREM 3.2. *Assume computation of* $\oplus$ *and* $\otimes$ *requires constant time. Extended distributivity is equivalent to the conditions for parallel tree contractions with respect to rose trees.*

Proof. [$\Leftarrow$] We are treating rose trees whose nodes have unbound degree. We assume leaves from outer positions into inner positions are gradually raked. A node with its value a can be regarded as $(a, \iota_\otimes, \iota_\otimes)$. Take its leftmost subtree. Let b be its computed value, and y the computed value of the remaining siblings appearing on its right. The reduced value of the subtrees under a is therefore $b \otimes y$. The rake operation is to merge $(a, \iota_\otimes, \iota_\otimes)$ with b, and returns $(a, b, \iota_\otimes)$ using associativity of the operator $\otimes$.

$$\begin{aligned}(a,\iota_\otimes,\iota_\otimes) \ominus (b \otimes y) &= a \oplus (\iota_\otimes(b \otimes y) \otimes \iota_\otimes) \\ &= a \oplus (b \otimes y \otimes \iota_\otimes) \\ &= (a,b,\iota_\otimes) \ominus y\end{aligned}$$

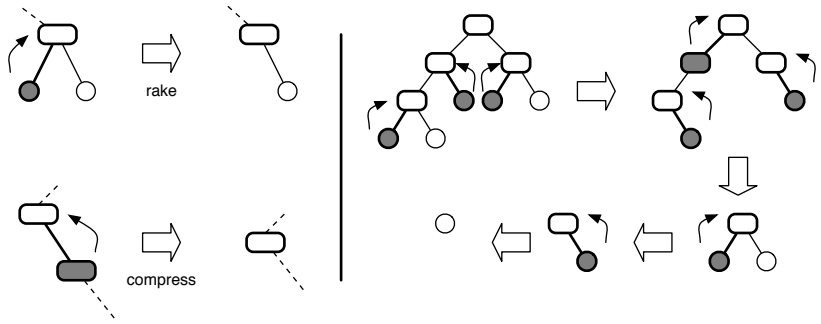


FIG. 3.2. *Tree contractions* rake *(upper left) and* compress *(lower left), and an example of parallel tree contractions (right)*

ROUTINE 4.1. *First phase applied to each fragment*

Input: Sequence $(d_0, a_0), \ldots, (d_{n-1}, a_{n-1})$ $(n \geq 1)$.
Variables: Arrays as, bs, cs (behaving as stacks growing from left to right), an integer d, and a value t.

(1) Set $d \leftarrow d_0$. If a_0 is "/" then $cs \leftarrow [\iota_\otimes, \iota_\otimes]$, $as \leftarrow [\,]$, $bs \leftarrow [\,]$; else $cs \leftarrow [\iota_\otimes]$, $as \leftarrow [a_0]$, $bs \leftarrow [\iota_\otimes]$.
(2) For each i in $\{1, \ldots, n-1\}$
  (2-a) if a_i is not "/" (namely a value), then push a_i to as and $\iota_\otimes$ to bs;
  (2-b) else if as is empty, then push $\iota_\otimes$ to cs, and set $d \leftarrow d_i$;
  (2-c) else pop a' from as and b' from bs.
    if $b' = \iota_\otimes$ (implying a' is a leaf) then $t \leftarrow h'(a')$; else $t \leftarrow a' \otimes b'$;
    if bs is not empty, then pop b'' from bs and push $b'' \otimes t$ to bs; else pop c'' from cs and push $c'' \otimes t$ to cs.
(3) If $\mathrm{P} \neq 1$ then remove $\iota_\otimes$ at the bottom of cs_0 and $cs_{\mathrm{P}-1}$.

In terms of the value c of its rightmost leaf, the rake operation similarly succeeds to produce $(a, \iota_\otimes, c)$. The same arguments apply to the cases in which there are already raked values, namely (a, b, c) in general.

The compress operation is to merge a node (a_u, b_u, c_u) with its subtree whose computed value can be written as $(a_l, b_l, c_l) \ominus e$. This is what extended distributivity deals with, and we successfully have $(a', b', c') = (a_u, b_u, c_u) \ominus ((a_l, b_l, c_l) \ominus e)$.

[⇒] When extended distributivity holds, we have the implementation of the tree homomorphism based on tree contractions over a binary tree representation of rose trees [27]. □

4. Parallelized Algorithm. This section develops a parallel algorithm for tree homomorphism which satisfies extended distributivity. Our algorithm consists of three phases: (1) the first phase applies tree homomorphism toward segments (consecutive subsequences) as much as possible which is distributed to each processor; (2) after communications among processors the second phase performs further reduction using extended distributivity, producing a binary tree as a result whose internal nodes are specified as *triple*s, and size is less than twice of the number of processors; finally (3) the third phase reduces the binary tree into a single value.

As was used in Introduction, we set to use N to denote the number of nodes in the tree we apply computations to, and P the number of available processors. The serialized representation has therefore 2N elements. During the explanation we assume $\mathrm{P} = 4$. Each processor is assumed to have $\mathrm{O}(\mathrm{N}/\mathrm{P})$ local memory, and to be connected by a router that can send messages in a point-to-point manner. Our algorithm developed here involves all-to-all transactions; such a mechanism is available in MPI on many PC clusters. We assume BSP model [43, 33].

4.1. First phase. Each processor applies tree homomorphism to its given segment of size $2\mathrm{N}/\mathrm{P}$. The process is summarized as Routine 4.1. This process leaves fragments of results, in arrays as_i, bs_i, cs_i and an integer d_i for each processor number i. The array as_i is to keep the open elements without their corresponding close element. Each element of as_i can have subtrees before the next one in as_i, and their reduced values are kept

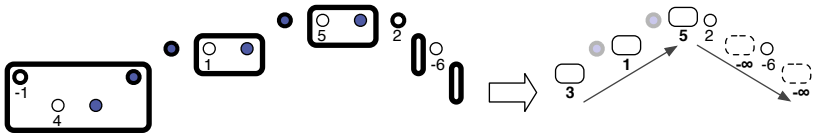
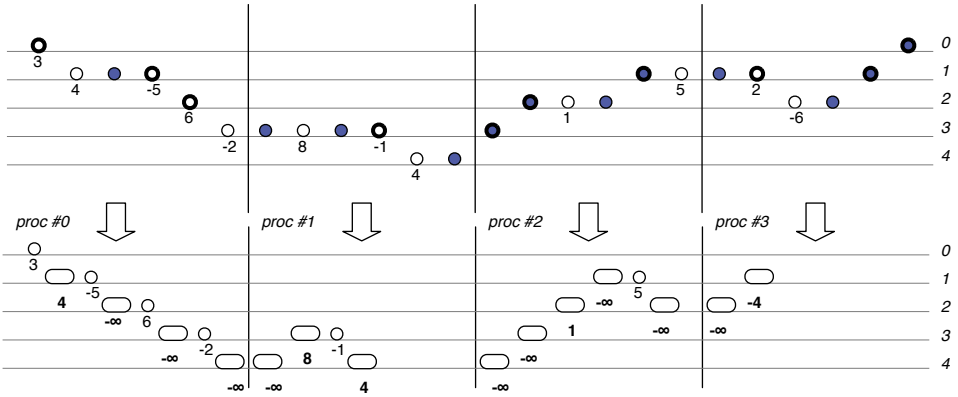


FIG. 4.1. *An illustrating example of the first phase—applying tree homomorphism maxPath to a segment of Fig. 2.2 (lined and dashed ovals indicate values obtained by reduction and $-\infty$ (the unit of $\uparrow$), respectively)*



$cs_0 = [\,]$	$cs_1 = [-\infty, 8]$	$cs_2 = [-\infty, -\infty, 1, -\infty]$	$cs_3 = [-\infty, -4]$
$as_0 = [3, -5, 6, 2]$	$as_1 = [-1]$	$as_2 = [5]$	$as_3 = [\,]$
$bs_0 = [4, -\infty, -\infty, -\infty]$	$bs_1 = [4]$	$bs_2 = [-\infty]$	$bs_2 = [\,]$
$d_0 = 0$	$d_1 = 3$	$d_2 = 1$	$d_3 = 0$

FIG. 4.2. *The results by First phase (upper: illustration, lower: data)*

in bs_i. Similar treatments are done to unmatched close elements, leaving values in cs_i (we remove unmatched close elements thanks to the absence of values). The integer d_i denotes the shallowest depth in processor i. While both of elements in as_i and bs_i are listed in a descending manner, those of cs_i are in ascending manner; the initial elements of as_i and bs_i and the last one of cs_i are at height d_i (except for cs_0 and $cs_{\mathrm{P}-1}$ whose last element at depth 0 is always $\iota_\otimes$ and therefore is set to be eliminated).

In Fig. 4.1 we show a case of the illustrating segment from the 10th element $(3, -1)$ to the 21st $(2, -6)$ of the sequence in Fig. 2.2. We have, as depicted:

$$\begin{array}{rclcrcl} cs & = & [-1 + id(4),\ id(1),\ id(5)] & & as & = & [2, -6], \qquad d = 1. \\ & = & [3, 1, 5], & & bs & = & [-\infty, -\infty], \end{array}$$

Please note that we regard absence of subtrees as an empty forest to which the tree homomorphism returns $-\infty$, the unit of $\uparrow$ (at depth 2 and 3 kept in bs). When we distribute the whole sequence in Fig. 2.2 evenly among four processors (6 elements for each), the results by this phase is shown in Fig. 4.2.

4.2. Second phase. The second phase matches data fragments kept in each processor into *triple*s (a, b, c) using communication between processors. Later, we reduce consecutive occurrences of *triple*s into a value, or into one *triple* by extended distributivity.

When we look carefully at Fig. 4.3, we notice that 3 in as_0 at depth 0 now has five parts at depth 1 as its children: the value 4 in bs_0, a subtree spanning from processors 0 to 2 whose root is -5 in as_0, the value $-\infty$ in cs_2, a subtree from processor 2 to 3 whose root is 5 in as_2, and the value -4 in cs_3. As these subtrees need reducing separately, we focus on the leftmost and the rightmost values in bs_0 and cs_3 (we leave the value $-\infty$ in cs_2 for the time being). We notice that the *group* of the value 3 in as_0 with these two values in processors 0 and 3 forms a *triple* $(3, 4, -4)$.

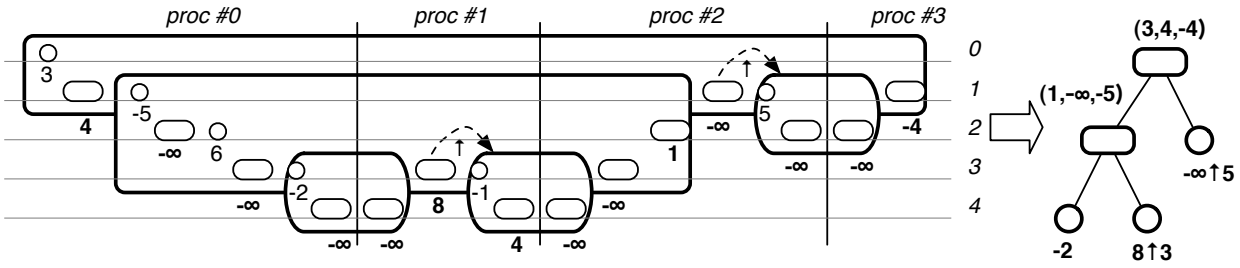


FIG. 4.3. Triple*s between two processors (left) and the resulting tree (right) after the second phase*

Similarly, two elements in as_0 at depth 1 and 2, with two elements each in bs_0 and cs_2 at depth 2 and 3, respectively, form two *triple*s $(-5, -\infty, 1)$ and $(6, -\infty, -\infty)$. The former *triple* indicates a tree that awaits the result of one subtree specified by the latter. This situation is what extended distributivity takes care of, and we can merge two *triple*s (a sequence of *triple*s in general) into one:

$$\begin{aligned}
&(-5, -\infty, 1) \ominus (\ (6, -\infty, -\infty) \ominus e\ ) \\
&\quad = \quad (\ (-5+6),\ (-6-\infty \uparrow -\infty),\ (-\infty \uparrow -6+1)\ ) \ominus e \\
&\quad = \quad (1, -\infty, -5) \ominus e
\end{aligned}$$

for any e. In this way, such *group*s of data fragments in two processors turn into one *triple*.

*Group*s from two adjacent processors are reduced into a single value without any missing subtrees in between. Instead of treating using extended distributivity, the values -2 in as_0 and -1 in as_1 at depth 3, and 5 in as_2 at depth 2 with their corresponding values in bs_i and cs_{i+1} $(i = 0, 1, 2)$ turn into values $id(-2) = -2$, $-1 + (4 \uparrow -\infty) = 3$, $id(5) = 5$, respectively.

We state the following lemma to tell the number of resulting *group*s in total.

LEMMA 4.1. *Given p processors. The second phase produces* group*s of the number at most* $2p - 3$.

Proof. For simplicity we first assume the shallowest depths d_i for $0 < i < p - 1$ are disjoint for each other (both d_0 and d_{p-1} are 0). Under this assumption the equality $R_p = 2p-3$ holds. Proof is given by mathematical induction.

Consider the case $p = 2$. We immediately see that $R_2 = 1 = 2 \cdot 2 - 3$. Assume $R_i = 2i - 3$ holds for $2 \le i < p$, and we have p processors each of which holds the results by Routine 4.1. First, we need to observe that $d_0 = d_{p-1} = 0$ and $d_0 < d_i$ for $0 < i < p - 1$, since we deal with well-formed trees only. By the assumption of disjoint d_i we can find $0 < i < p$ such that $d_i > d_j$ for $0 < j < p$, $i \neq j$. Using the height d_i, we find a *group* between processors 0 and p; we continue investigation of *group*s for each of $i + 1$ processors (from 0 to i) and $p - i$ processors (from i to $p - 1$) with their initial height d_i instead of 0. Hence the following holds, using induction hypothesis.

$$\begin{aligned}
R_p &= 1 + R_{i+1} + R_{p-1-i+1} \\
&= 1 + (2(i+1) - 3) + (2(p-i) - 3) \\
&= 2p - 3
\end{aligned}$$

Inequality appears in case there appear the same shallowest depths. For example, assume there are four processors, and processors 1 and 2 have the same peak depth, namely $d_1 = d_2 > d_0 = d_3 = 0$. In this case, the *group*s are created between processors 0 and 3, 0 and 1, 1 and 2, 2 and 3. This partitioning produces 4 *group*s, one smaller than $5 = 2 \cdot 4 - 3$. Generalization of this argument proves $R_P \le 2P - 3$. □

This lemma guarantees that, the number of *group*s this phase produces is less than twice of the number of processors. Notice that the *group*s form a tree (Fig. 4.3, right). This observation enables us to have another explanation on the number of *group*s. Given p processors, we soon notice that there exist *group*s between two adjacent processors. They are regarded as leaves, without any missing subtrees in between. The number of

ROUTINE 4.2. *Second phase computing groups among processors.*

Input: Sequence (p, d_p) is given in the ascending order of p.
Variables: A stack is used whose top is referred as (p_s, d_s).

(1) Push the first pair $(0, 0)$ on a stack
(2) For each i in $\{1, \ldots, \mathrm{P} - 1\}$
  (2-1) prepare a variable $d \leftarrow \infty$.
  (2-2) while $d_i < d_s$, produce $M^{p_s \leftrightarrow i}_{[d_s, d]}$, set $d \leftarrow d_s$ and pop from the stack;
  (2-3) if $d_i = d_s$, then produce $M^{p_s \leftrightarrow i}_{[d_s, d]}$ and $M^{\leftrightarrow}_{[d_i, d_s]}$, and pop from the stack; else produce $M^{p_s \leftrightarrow i}_{[d_i, d]}$.
  (2-4) push (i, d_i).
(3) Finally eliminate the last mating pair (that is $M^{\leftrightarrow}_{[0,0]}$).

leaves are therefore one smaller than the processor numbers, namely $p - 1$. Other *group*s behave as *internal nodes* which have two or more subtrees (otherwise we can simplify two *group*s of direct filiation into one *group* using extended distributivity). In case the number of leaves are fixed, binary trees have the largest number of internal nodes among trees in general. The number of internal nodes in a binary tree is one smaller than that of leaves, and if there are $p - 1$ leaves we have $p - 2$ internal nodes. Hence the number of *group*s is no more than $2p - 3$.

The Routine 4.2 figures out *group*s among processors. $M^{p_l \leftrightarrow p_r}_{[d_u, d_l]}$ denotes a *group* between processors p_l and p_r whose data fragments span from the depth d_u until d_l (∞ in d_l indicates "everything starting from d_u"). This Routine inserts $M^{\leftrightarrow}_{[d_u, d_l]}$ as a dummy *group* in case the same d appear among more than two consecutive processors. It is assumed to be reduced into $\iota_\ominus$, a virtual left unit of $\ominus$ (namely $\iota_\ominus \ominus e = e$). This routine produces $2\mathrm{P} - 3$ *group*s in the *post-order* traversal over the binary tree.

The remaining task in this phase is to perform data transactions of as_i, bs_i, cs_i among processors according to the *group*s information $M^{p_l \leftrightarrow p_r}_{[d_u, d_l]}$, and apply further computation toward each *group*. There can be a couple of approaches of data transaction. One simple idea is to transfer fragments of cs_i to their corresponding processors. When we apply computation for each *group*, the computed value at the shallowest depth d_p in each processor are associated to the *group* on their right for later computation by $\otimes$ (8 in cs_1, $-\infty$ in cs_2; see Fig. 4.3).

4.3. Third phase. This last phase compiles obtained *triple*s or values and reduce them into a single value. As Fig. 4.3 shows, the obtained *triple*s and values in the previous phase form a binary tree of size $2\mathrm{P} - 3$ (including dummies by $M^{\leftrightarrow}_{[0,0]}$). We collect *triple*s or values in one processor, and apply tree reduction in $\mathrm{O}(\mathrm{P})$ time.

4.4. Cost estimation. The whole procedures are summarized as Algorithm 4.1. As the summary of this section, we estimate the cost of this algorithm.

C_h, C_{ed}, and C_t are, respectively, the computational cost of tree homomorphism (using h', $\otimes$ and $\oplus$), that of merging two *triple*s into one by extended distributivity using its characteristic functions, and that of *group*s generation by Routine 4.2. The length of an array xs is written as $|xs|$. The cost ratio of communication compared to computation is written as g, and L is the time required for barrier synchronization among all processors.

The cost is summarized in Table 4.1. Step (1) takes time linear to the data size in each processor. The memory requirement of this sequential procedure (Routine 4.1) is $\mathrm{O}(\mathrm{N}/\mathrm{P})$ as well. The worst case in terms of the size of the results occurs when an sequence of only open (or close) elements is given, resulting in two arrays as_i and bs_i (or cs_i) of the length $2\mathrm{N}/\mathrm{P}$ for each. Step (2-2) takes $\mathrm{O}(\mathrm{P})$ time and space.

Step (2-3) requires detailed analysis. The cost depends on how each processor sends out its fragment of data. This depends on the size of cs_i, namely $|cs_i|$. When we analyze the worst case, it is possible that a processor sends out all of its cs_i whose size can be at most $2\mathrm{N}/\mathrm{P}$. Therefore the worst cost is estimated as $g \cdot 2\mathrm{N}/\mathrm{P}$, and this is $\mathrm{O}(\mathrm{N}/\mathrm{P})$.

Similar analysis applies to Step (2-4), by changing the viewpoint from the transmitter to the receiver. It is often the case that the computational cost C_{ed} is heavier than C_h. The worst case occurs when the length of

ALGORITHM 4.1. *The whole procedure of tree reduction.*

Input: Assume the serialized representation of a tree of size N (the length of the list is 2N) is partitioned into P sublists, which are distributed among processors $0, \ldots, P-1$.

First phase:

(1) Each processor sequentially performs tree computation using Routine 4.1, and produces arrays as_i, bs_i, cs_i and the shallowest depth d_i .

Second phase:

(2-1) All values of d_i are shared through global communication using all-to-all transactions.

(2-2) Each processor performs Routine 4.2 to figure out the structures *group*s have.

(2-3) Each processor transmits fragments of cs_i to their corresponding processor according to the information obtained in Step (2-2).

(2-4) Each processor reduces *group*s into single values or single *triple*s.

Third phase:

(3-1) Values and *triple*s obtained in Step (2-4) are collected to processor 0.

(3-2) Processor 0 reduces the binary tree into the result.

TABLE 4.1
Cost estimation of our algorithm under BSP

Step	computation	communication	synch.
(1)	$C_1 \cdot 2N/P$		
(2-1)		$g \cdot P$	L
(2-2)	$C_3 \cdot P$		
(2-3)		$\max_i\{g \cdot \lvert cs_i\rvert\}$	L
(2-4)	$\leq\ C_{ed} \cdot \max_i\{\lvert as_i\rvert\}$		
(3-1)		$\leq\ g \cdot P$	L
(3-2)	$C_h \cdot (\ (2P-3)-1\ )$		

as_i is $2N/P$, and when the *group*s in that processor behave as internal nodes, requiring computation of extended distributivity.

After Step (2-4) we have a binary tree of size $2P-3$ whose nodes are distributed among P processors. The processor 0 collects these results and applies the final final reduction. The cost for the final computation (3-2) is therefore $O(P)$.

We conclude this section by stating the following theorem.

THEOREM 4.2. *Tree homomorphism with extended distributivity has a BSP implementation with three supersteps of at most* $O(P + N/P)$ *communication cost for each.*

5. Experiments. We performed experiments using our implementation using C++ and MPI. The environment we used was semi-uniform PC clusters which consist of 2.4GHz or 2.8GHz processors with 2GB memory each and are connected with Gigabit Ethernet. The compiler and MPI library are gcc 4.1.1 and MPICH 1.2.7. We tested the efficacy of our algorithm using two kinds of experiments. The first is a querying operations over given trees. These computations are specified as a tree homomorphism using operations over matrices of size 10×10. The second is what we have seen as the running example, namely *maxPath*. The former examines the cases where heavy computation is involved, while the latter with simple computations will exhibit the communication costs which our algorithm introduces. We prepared trees of size 1,000,000 in three types, namely (F) a flat tree, (M) a monadic tree, and (R) 10 examples of randomly generated trees. A flat tree of size n is a tree with $n-1$ leaves just below the root. A monadic tree is a tree-view of a list, and has internal nodes with just one subtree for each. We executed the program over each type using 2^i processors ($i = 0, \ldots, 6$). We

TABLE 5.1
Execution times of the first experiments (querying, time in seconds)

	Random Trees (R)				Flat (F)		Monadic (M)	
p	dist.	comp.	min.	max.	dist.	comp.	dist.	comp.
1	0.088	5.546	5.494	5.613	0.090	3.833	0.109	11.924
2	0.210	2.814	2.753	2.892	0.208	1.886	N.A.	N.A.
4	0.252	1.493	1.469	1.527	0.250	1.037	0.248	12.474
8	0.308	0.751	0.736	0.762	0.301	0.529	0.301	6.810
16	0.339	0.377	0.367	0.381	0.354	0.287	0.332	3.939
32	0.375	0.191	0.188	0.196	0.366	0.138	0.395	2.057
64	0.447	0.101	0.097	0.105	0.459	0.070	0.422	1.587

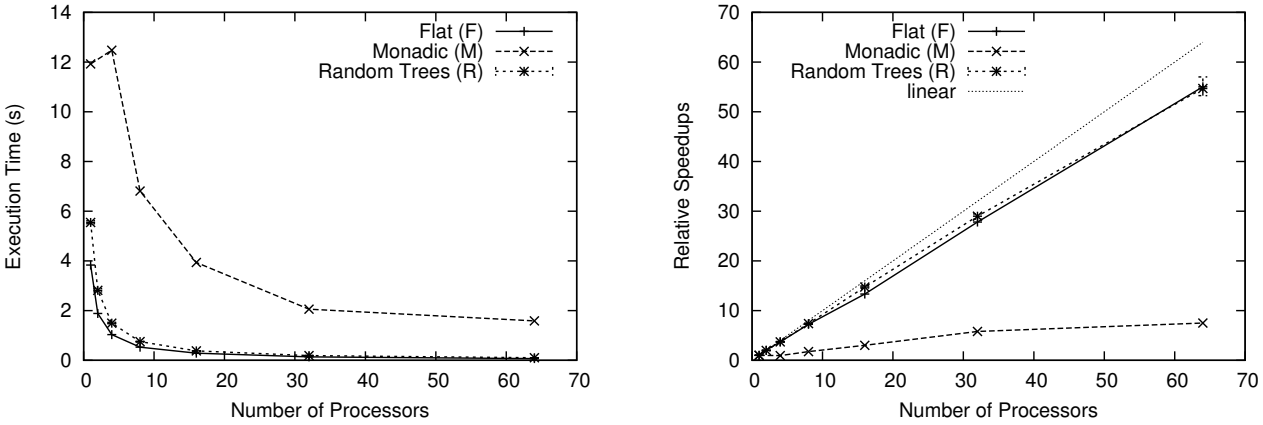


FIG. 5.1. *Plots of Table 5.1 by total execution time (left) and by speedups of computation time (right)*

repeated the same experiments five times, and their average times of the initial data distribution and of the computation itself ("dist." and "comp.", respectively) are summarized in the following tables (measurement in second). Randomly shaped trees (R) has various shapes and their computation time can naturally vary. The minimum and maximum are also put in Tables ("min." and "max.", respectively).

Table 5.1 shows the results of the first experiments of querying operations. As their plots in the left of Fig. 5.1 indicates, our algorithm exhibited good scalability. Results of (M) fell behind the other two. We can point out three reasons. The first reason is failure to use caches effectively. Routine 1 in (1) which uses arrays in a stack-like manner can enjoy caching effects when the corresponding open and close elements are located closely. The serialized representation of monadic trees is a sequence of open elements and a sequence of close elements afterwards. This hampers locality, and we can observe the penalties of high cache misses under a single processor $\mathrm{P} = 1$ where no parallelization is taken place. The second reason is communication and computation cost in Steps (2-3) and (2-4). We cannot apply any computation with monadic trees at (1), and this step has to leave as_i, bs_i and cs_j intact with their length $2\mathrm{N}/\mathrm{P}$ ($0 \leq i < \mathrm{P}/2$, $\mathrm{P}/2 \leq j < \mathrm{P}$). The third reason is communication anomalies when larger data are passed at Step (2-3). We will analyze this using the next experiments.

The results of the second experiments using our running example *maxPath* appear in Table 5.2. Since the required computations are quite cheap, improvements by parallelization are limited, and it does not pay off to perform parallelization for this data size. Instead, these experiments exhibit the cost of parallelization, especially that of data transactions. We make two notes on anomalies of communications. The first is that the variance of execution time becomes large as the number of processors increases. We observed that there were at large 10 msecs difference in execution time under $\mathrm{P} = 64$. The flat trees has least communication and computation cost, but there appears inversion phenomena that our algorithm ran faster over random trees than over a flat tree ($\mathrm{P} = 4, 8, 16$). The second is about congestion of network, which apparently happens toward the

TABLE 5.2
Execution times of the second experiments (maxPath, time in second)

	Random Trees (R)				Flat (F)		Monadic (M)	
p	dist.	comp.	min.	max.	dist.	comp.	dist.	comp.
1	0.088	0.055	0.054	0.058	0.086	0.044	0.086	0.086
2	0.207	0.028	0.027	0.032	0.226	0.022	0.210	0.170
4	0.252	0.016	0.014	0.019	0.248	0.018	0.250	0.086
8	0.309	0.013	0.007	0.016	0.306	0.016	0.301	0.049
16	0.342	0.010	0.005	0.012	0.355	0.011	0.333	0.031
32	0.375	0.013	0.004	0.028	0.397	0.011	0.366	0.022
64	0.450	0.019	0.007	0.033	0.456	0.007	0.490	0.139

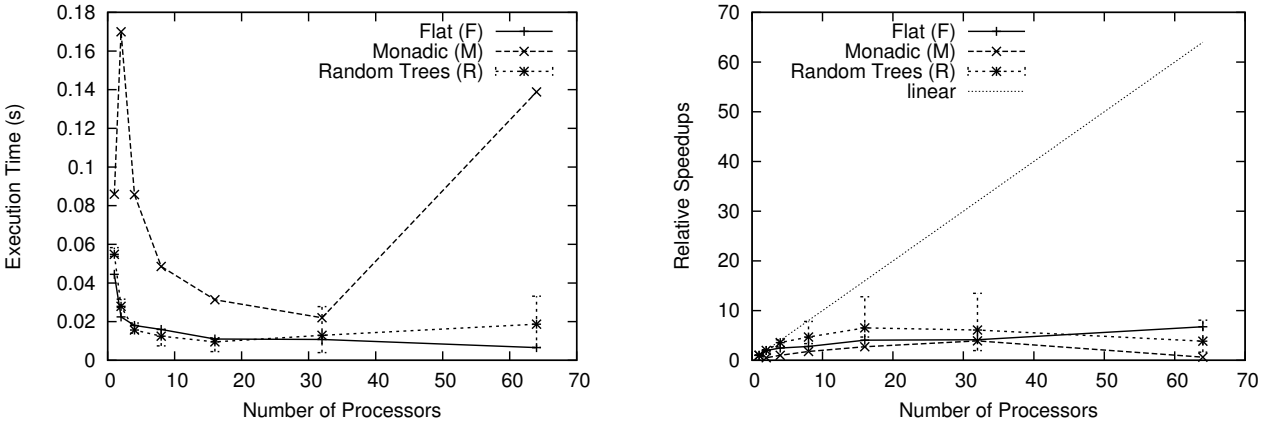


FIG. 5.2. *Plots of Table 5.2 by total execution time (left) and by speedups of computation time (right)*

monadic tree under $P = 64$. Monadic trees require Step (2-3) to transmit data of large size, but the amount of whole transmitted data through network does not change as $\sum_i |cs_i| = 2N/P + O(P)$ for any $P \geq 2$. Our algorithm with all-to-all transactions has to have vulnerability to the configuration and status of networks.

As a final note, it is natural that no difference existed in terms of costs of initial data distribution regardless of the shape of the trees.

6. Related Work. This section compares our proposed framework with related work. It mainly spans parallel tree contractions and list ranking algorithms, flattening transformation, parentheses matching, list homomorphism, and MapReduce-based implementation.

6.1. Parallel tree contractions. The parallel tree contractions algorithm, first proposed by Miller and Reif [34], are very important parallel algorithms for trees. Many researchers have devoted themselves to developing efficient implementations on various parallel models [18, 14, 1, 3, 31, 32, 16, 2]. Among researches based on shared memory environments, Gibbons and Rytter developed an optimal algorithm on CREW PRAM [18]; Abrahamson et al. developed an optimal and efficient algorithm in $O(N/P + \log P)$ on EREW PRAM [1].

Recently parallel tree contractions as well as list ranking, which serves the basis of parallel tree contractions, have been analyzed under the assumption of distributed memory environments. Mayr and Werchner showed $O(\lceil N/P \rceil \log P)$ implementations on hypercubes or related hypercubic networks [31, 32]. Dehne et al. solved list ranking and tree contractions on CGM/BSP using $O(N/P \log P)$ parallel time [16]. Sibeyn proposed a list ranking algorithm which aimed at reduction of communication costs [40].

Our refined algorithm runs in $O(N/P + P)$ and is much improved result which comes close to the algorithms under PRAM model. The advantage of our algorithm is not limited to the theoretical aspect. As our experiments demonstrated, the data representation we employed suits current computer architectures which extensively relies on caching mechanism for fast execution. Linked structures, namely dynamic data structures, involve pointers

and their data fragments can scatter over the memory heap. Flexible though they are in terms of structure manipulation like insertion and deletion, scattered data can easily lack locality.

The second advantage is the cost and concerns of data distribution. In case we have sophisticated distribution approaches of tree structures we can employ the EREW-PRAM parallel tree contraction algorithm. One such technique is based on m-bridges [38]. This technique translates a binary tree into another binary tree in each of whose nodes locates a subtrees of the original tree partitioned into almost the same size. Our group has another implementation for parallel tree computation based on this approach [41, 29, 28, 26]. The drawback of the approach using m-bridge is its cost. When trees are kept distributively among processors, the algorithm for realizing m-bridges requires Euler tour and list ranking, which as a result spoils theoretical complexity and running time in total. It should be noted that the parallelized algorithm presented in this paper runs in $O(N/P + P)$. This cost is theoretically comparable to the EREW-PRAM parallel tree contractions algorithm, using ignorable cost and troubles of initial data distribution.

6.2. Nested parallelism and flattening transformation. Blelloch's *nested parallelism* and the language NESL addresses the importance of data-parallel computation toward nested structures (often in the form of nested lists), where the length of each list can differ [6]. The idea proposed is *flattening* of the structures [7]. The importance of this problem domain ignited a lot of researches afterwards, theoretically and in real compilers [8, 39, 10].

Our idea presented in this paper is one instance of flattening transformation where segment descriptors are diffused into the flattened data. To cope with irregularity of tree structures we format trees into its serialized representation with an additional tags indicating the depth. The flattening of tree structures has already been researched. Prins and Palmer developed data-parallel language Proteus, and nesting trees were treated [37, 35]. Chakravarty and Keller extended the structure to involve trees and recursive data structures in general [24, 9]. Their computation framework, however, stayed to treat horizontal computation in parallel. As far as we are aware, this paper is the first to relate flattening transformations with the parallel tree contractions to derive parallelism in the vertical direction.

6.3. Parentheses matching. The process of our algorithm development much resembles parentheses matching algorithms, or the All Nearest Smaller Values Problems (often called ANSV for short). Their algorithms have been analyzed under CRCW PRAM [4], EREW PRAM [36], hypercube [25] and BSP [21]. The resemblance naturally comes from how tree structures are translated in sequence; the generation of *group*s in Routine 2 follows the process to find matching of ANSV developed in [4]. We have made a step forward to apply computation over the data structures. The implementation under BSP has complexity of $O(N/P + P)$. Our algorithm naturally has complexity comparable to it.

He and Huang analyzed that the cost of data transaction becomes constant toward sequences of random values [21]. Unfortunately this observation does not hold when we are to compute tree reductions. The first reason is that we have to transmit not only the information of shallowest depth d_i, but the computed results cs_i to realize reduction computations. Secondly, their style of analysis based on the assumption of random input does not apply since the serialized representation of trees has certain properties. For example, given a sequence $A = a_0, a_1, \ldots, a_{2n-1}$ which consists of open and close elements. In order to be a well-formed serialized representation, A has exactly n open and n close elements, and the number of open elements in any subsequence of A, namely $a_0, \ldots a_i$ $(i < 2n)$, has to be no less than that of close elements. While we haven't developed qualitative analyses so far, the experiments using randomly generated trees in this paper indicates that the communication cost stays in a reasonable amount.

6.4. List homomorphism. List homomorphism is a model that plays an important role for developing efficient parallel programs [13, 19, 20, 22]. A function h^{L} is a homomorphism if there exist an associative operator $\oplus$ and a unary function g such that

$$\begin{aligned} h^{\mathrm{L}}(x +\!\!+ y) &= h^{\mathrm{L}}\ x \odot h^{\mathrm{L}}\ y\ , \\ h^{\mathrm{L}}\ [a] &= g\ a\ . \end{aligned}$$

This function can be efficiently implemented in parallel since it ideally suits for divide-and-conquer, bottom-up computation: a list is divided into two fragments x and y recursively, and the computations of $h^{\mathrm{L}}\ x$ and

$h^{\mathrm{L}}\ y$ can be carried out in parallel. For instance, the function *sum*, which computes the sum of all the elements in a list, is a homomorphism because the equations $sum\ (x +\!\!+ y) = sum\ x + sum\ y$ and $sum\ [a] = a$ hold. This indicates that we can reduce parallel programming into construction of list homomorphism.

In further detail, parallel environments for distributed lists evaluate list homomorphism in two phases after distribution of data: (a) the local computation at each processor (using g and $\odot$), and (b) the global computation among processors (using $\odot$). Assume the computation of g and $\odot$ requires constant time. By evenly distributing consecutive elements to processors, the phase (a) takes $\mathrm{O}(\mathrm{N}/\mathrm{P})$ cost. The phase (b) reduces these P values into a single value tree-recursively in $\log \mathrm{P}$ iterations. The total cost is therefore estimated as $\mathrm{O}(\mathrm{N}/\mathrm{P} + \log \mathrm{P})$.

Tree-recursive data communication, which is the basis of efficient execution for list homomorphism, however, seems restrictive for this problem. Its resulting complexity is, even with nested use of homomorphism, $\mathrm{O}(\log^2 \mathrm{N})$ for abundance of processors. We made a test implementation for this approach, and we observed the scalability was tamed much earlier: the speedup ratios from $\mathrm{P} = 16$ to 32, and from 32 to 64 using the first experiments of querying were, respectively, 1.81 and 1.26, and they did not catch up the respective ratios 1.97 and 1.89 obtained by the approach in this paper.

6.5. MapReduce-based Implementation. MapReduce is a framework for large-scale data processing proposed by Google [15], and its open-source implementation in Hadoop [44] is now widely used. Though the conventional programming model of MapReduce is for unordered data (sets), with some extension in Hadoop MapReduce we can deal with ordered data including serialized representation of trees. Recently, dealing XML documents on MapReduce has been studied actively. For example, Choi at al. developed an XML querying system on Hadoop MapReduce [11]. They also proposed algorithms for labeling XML trees with MapReduce [12]. It is worth noting that the idea similar to parentheses matching was used in the latter to process unmatching tags in XML fragments.

The serialized representation of trees and flexibility of partitioning in our approach is also suitable for the implementation of tree manipulations on MapReduce. Based on the earlier version of this work, Emoto and Imachi developed a MapReduce algorithm for tree reductions [17]. Furthermore, the algorithm was extended to tree accumulations by Matsuzaki and Miyazaki [30], which can be implemented with two-round MapReduce computation.

7. Concluding Remarks. In this paper we have developed a new approach for parallel tree reductions. The essence of our approach is to make good use of the serialized representation of trees in terms of initial data distribution and computation using high memory locality. The results by the local computation forms a binary tree of size less than twice of the number of processors and each node has size $\mathrm{O}(\mathrm{N}/\mathrm{P})$. Our algorithm has a BSP implementation with three supersteps. Our test implementation showed good scalability in general, but we also observed network fluctuation under PC clusters.

This research will be improved in the following aspects. Our experiments exhibited good performance toward random inputs. It is an interesting mathematical question how much amount of transactions and computation we have to execute in Steps (2-3) and (2-4) in average. The flexibility in partitioning our representations will be beneficial under heterogeneous environments with different processor ability. After the process of *group* generation, data can be reallocated among processors, regardless of their computational power. Theoretical support for these issues are needed.

Acknowledgment. This research was partially supported by the Ministry of Education, Culture, Sports, Science and Technology, Grant-in-Aid for Young Scientists (B), 17700026, 2005–2007.

REFERENCES

[1] K. R. Abrahamson, N. Dadoun, D. G. Kirkpatrick, and T. M. Przytycka, *A simple parallel tree contraction algorithm*, J. Algorithms, 10 (1989), pp. 287–302.

[2] D. A. Bader, S. Sreshta, and 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, Bangalore, India, December 18-21, 2002, Proceedings, S. Sahni, V. K. Prasanna, and U. Shukla, eds., vol. 2552 of Lecture Notes in Computer Science, Springer, 2002, pp. 63–78.

[3] R. P. K. Banerjee, V. Goel, and A. Mukherjee, *Efficient parallel evaluation of CSG tree using fixed number of processors*, in Solid Modeling and Applications, 1993, pp. 137–146.
[4] O. Berkman, B. Schieber, and U. Vishkin, *Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values*, J. Algorithms, 14 (1993), pp. 344–370.
[5] G. E. Blelloch, *Scans as primitive parallel operations*, IEEE Trans. Computers, 38 (1989), pp. 1526–1538.
[6] ———, *Programming parallel algorithms*, Commun. ACM, 39 (1996), pp. 85–97.
[7] G. E. Blelloch and G. Sabot, *Compiling collection-oriented languages onto massively parallel computers*, J. Parallel Distrib. Comput., 8 (1990), pp. 119–134.
[8] D. C. Cann, *Retire fortran? A debate rekindled*, Commun. ACM, 35 (1992), pp. 81–89.
[9] M. M. T. Chakravarty and G. Keller, *More types for nested data parallel programming*, in Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000., M. Odersky and P. Wadler, eds., ACM, 2000, pp. 94–105.
[10] M. M. T. Chakravarty, G. Keller, R. Lechtchinsky, and W. Pfannenstiel, *Nepal — nested data parallelism in Haskell*, in Euro-Par 2001: Parallel Processing, 7th International Euro-Par Conference Manchester, UK August 28-31, 2001, Proceedings, R. Sakellariou, J. A. Keane, J. R. Gurd, and L. Freeman, eds., vol. 2150 of Lecture Notes in Computer Science, Springer, 2001, pp. 524–534.
[11] H. Choi, K.-H. Lee, S.-H. Kim, Y.-J. Lee, and B. Moon, *HadoopXML: A suite for parallel processing of massive XML data with multiple twig pattern queries*, in Proceedings of the 21st ACM International Conference on Information and Knowledge Management (CIKM'12), ACM, 2012, pp. 2737–2739.
[12] H. Choi, K.-H. Lee, and Y.-J. Lee, *Parallel labeling of massive XML data with MapReduce*, Journal of Supercomputing, 67 (2014), pp. 408–437.
[13] M. Cole, *Parallel programming with list homomorphisms*, Parallel Processing Letters, 5 (1995), pp. 191–203.
[14] R. Cole and U. Vishkin, *Optimal parallel algorithms for expression tree evaluation and list ranking*, in VLSI Algorithms and Architectures, 3rd Aegean Workshop on Computing, AWOC 88, Corfu, Greece, June 28 - July 1, 1988, Proceedings, J. H. Reif, ed., vol. 319 of Lecture Notes in Computer Science, Springer, 1988, pp. 91–100.
[15] J. Dean and S. Ghemawat, *MapReduce: Simplified data processing on large clusters*, in 6th Symposium on Operating System Design and Implementation (OSDI2004), December 6–8, 2004, San Francisco, California, USA, 2004, pp. 137–150.
[16] F. K. H. A. Dehne, A. Ferreira, E. Cáceres, S. W. Song, and A. Roncato, *Efficient parallel graph algorithms for coarse-grained multicomputers and BSP*, Algorithmica, 33 (2002), pp. 183–200.
[17] K. Emoto and H. Imachi, *Parallel tree reduction on MapReduce*, in Proceedings of the International Conference on Computational Science (ICCS 2012), vol. 9 of Procedia Computer Science, Elsevier, 2012, pp. 1827–1836.
[18] A. Gibbons and W. Rytter, *An optimal parallel algorithm for dynamic expression evaluation and its applications*, in Foundations of Software Technology and Theoretical Computer Science, Sixth Conference, New Delhi, India, December 18-20, 1986, Proceedings, K. V. Nori, ed., vol. 241 of Lecture Notes in Computer Science, Springer, 1986, pp. 453–469.
[19] S. Gorlatch, *Constructing list homomorphisms*, Tech. Report MIP-9512, Fakultät für Mathematik und Informatik, Universität Passau, August 1995.
[20] Z. N. Grant-Duff and P. G. Harrison, *Parallelism via homomorphisms*, Parallel Processing Letters, 6 (1996), pp. 279–295.
[21] X. He and C. Huang, *Communication efficient BSP algorithm for all nearest smaller values problem*, J. Parallel Distrib. Comput., 61 (2001), pp. 1425–1438.
[22] Z. Hu, H. Iwasaki, and M. Takeichi, *Formal derivation of efficient parallel programs by construction of list homomorphisms*, ACM Trans. Program. Lang. Syst., 19 (1997), pp. 444–461.
[23] K. Kakehi, K. Matsuzaki, and K. Emoto, *Efficient parallel tree reductions on distributed memory environments*, in Computational Science - ICCS 2007, 7th International Conference, Beijing, China, May 27 - 30, 2007, Proceedings, Part II, Y. Shi, G. D. van Albada, J. Dongarra, and P. M. A. Sloot, eds., vol. 4488 of Lecture Notes in Computer Science, Springer, 2007, pp. 601–608.
[24] G. Keller and M. M. T. Chakravarty, *Flattening trees*, in Euro-Par '98 Parallel Processing, 4th International Euro-Par Conference, Southampton, UK, September 1-4, 1998, Proceedings, D. J. Pritchard and J. Reeve, eds., vol. 1470 of Lecture Notes in Computer Science, Springer, 1998, pp. 709–719.
[25] D. Kravets and C. G. Plaxton, *All nearest smaller values on the hypercube*, IEEE Trans. Parallel Distrib. Syst., 7 (1996), pp. 456–462.
[26] K. Matsuzaki, *Efficient implementation of tree accumulations on distributed-memory parallel computers*, in Computational Science - ICCS 2007, 7th International Conference, Beijing, China, May 27 - 30, 2007, Proceedings, Part II, Y. Shi, G. D. van Albada, J. Dongarra, and P. M. A. Sloot, eds., vol. 4488 of Lecture Notes in Computer Science, Springer, 2007, pp. 609–616.
[27] K. Matsuzaki, Z. Hu, K. Kakehi, and M. Takeichi, *Systematic derivation of tree contraction algorithms*, Parallel Processing Letters, 15 (2005), pp. 321–336.
[28] K. Matsuzaki, Z. Hu, and M. Takeichi, *Parallel skeletons for manipulating general trees*, Parallel Computing, 32 (2006), pp. 590–603.
[29] K. Matsuzaki, H. Iwasaki, K. Emoto, and 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, Hong Kong, May 30-June 1, 2006, X. Jia, ed., vol. 152 of ACM International Conference Proceeding Series, ACM, 2006, p. 13.
[30] K. Matsuzaki and R. Miyazaki, *Parallel tree accumulations on MapReduce*, International Journal of Parallel Programming, 44 (2016), pp. 466–485.
[31] E. W. Mayr and R. Werchner, *Optimal routing of parentheses on the hypercube*, J. Parallel Distrib. Comput., 26 (1995), pp. 181–192.

[32] ———, *Optimal tree contraction and term matching on the hypercube and related networks*, Algorithmica, 18 (1997), pp. 445–460.
[33] W. F. McColl, *Scalable computing*, in Computer Science Today: Recent Trends and Developments, J. van Leeuwen, ed., vol. 1000 of Lecture Notes in Computer Science, Springer, 1995, pp. 46–61.
[34] G. L. Miller and J. H. Reif, *Parallel tree contraction and its application*, in 26th Annual Symposium on Foundations of Computer Science, Portland, Oregon, USA, 21-23 October 1985, IEEE Computer Society, 1985, pp. 478–489.
[35] D. W. Palmer, J. F. Prins, and S. Westfold, *Work-efficient nested data-parallelism*, in Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation (Frontiers'95), FRONTIERS '95, Washington, DC, USA, 1995, IEEE Computer Society, pp. 186–.
[36] S. K. Prasad, S. K. Das, and C. C. Chen, *Efficient EREW PRAM algorithms for parentheses-matching*, IEEE Trans. Parallel Distrib. Syst., 5 (1994), pp. 995–1008.
[37] J. Prins and D. W. Palmer, *Transforming high-level data-parallel programs into vector operations*, in Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP), San Diego, California, USA, May 19-22, 1993, M. C. Chen and R. Halstead, eds., ACM, 1993, pp. 119–128.
[38] M. Reid-Miller, G. L. Miller, and F. Modugno, *List ranking and parallel tree contraction*, in Synthesis of Parallel Algorithms, J. Reif, ed., Morgan Kaufmann, 1993, ch. 3, pp. 115–194.
[39] J. Riely and J. Prins, *Flattening is an improvement*, in Static Analysis, 7th International Symposium, SAS 2000, Santa Barbara, CA, USA, June 29 - July 1, 2000, Proceedings, J. Palsberg, ed., vol. 1824 of Lecture Notes in Computer Science, Springer, 2000, pp. 360–376.
[40] J. F. Sibeyn, *One-by-one cleaning for practical parallel list ranking*, Algorithmica, 32 (2002), pp. 345–363.
[41] *Sketo project home page*. `http://sketo.ipl-lab.org/`.
[42] D. B. Skillicorn, *Parallel implementation of tree skeletons*, J. Parallel Distrib. Comput., 39 (1996), pp. 115–125.
[43] L. G. Valiant, *A bridging model for parallel computation*, Commun. ACM, 33 (1990), pp. 103–111.
[44] T. White, *Hadoop: The Definitive Guide*, O'Reilly Media / Yahoo Press, 2012.

Edited by: Frédéric Loulergue
Received: September 16, 2016
Accepted: January 17, 2017