MATHEMATICAL ENGINEERING
TECHNICAL REPORTS

A Practicable Framework for Tree Reductions under Distributed Memory Environments

Kazuhiko KAKEHI, Kiminori MATSUZAKI,
Kento EMOTO, and Zhenjiang HU

METR 2006–64 December 2006

DEPARTMENT OF MATHEMATICAL INFORMATICS
GRADUATE SCHOOL OF INFORMATION SCIENCE AND TECHNOLOGY
THE UNIVERSITY OF TOKYO
BUNKYO-KU, TOKYO 113-8656, JAPAN

WWW page: http://www.i.u-tokyo.ac.jp/mi/mi-e.htm

A Practicable Framework for Tree Reductions under Distributed Memory Environments

Kazuhiko Kakehi[*1] Kiminori Matsuzaki[*2]
Kento Emoto[*3] Zhenjiang Hu[*2]

[*1]Division of University Corporate Relations
[*2]Department of Mathematical Informatics
[*3]Department of Creative Informatics
University of Tokyo
Hongo, Tokyo 113–0033, Japan
Email: {kaz,kmatsu,emoto,hu}@ipl.t.u-tokyo.ac.jp

Abstract

Besides intensive research toward matrices or one dimensional arrays, another important data structure, namely trees, are calling for efficient parallel treatments. Parallel tree contractions are fundamental to realize efficient computation over the inherently imbalanced structures. However, we sometimes fail to practically benefit from the techniques under current computer architectures. This is because data representations matter not only to distributed memory environments under which data is kept separately among processors, but also to current processor designs on which linked structures gain poorly. The assumption of linked structures spoils parallel complexity as well.

This paper proposes a new approach for fast parallel computation on trees over distributed memory environments. The characteristics of our approach is the internal data representation. We employ the serialized forms of trees obtained by tree traversals. These representations benefit from ease of data distribution and current processor designs. We present two approaches for parallelizing tree reduction computations.

The first algorithm is designed straightforwardly based on the list homomorphic approach. It has good scalability in many cases, yet its complexity depends on the factor of tree depth. To perform improvement, the second algorithm introduces in the tree reductions an additional condition, which is equivalent to that of parallel tree contractions. By exploiting parallelizable parts further, this refined algorithm is implemented under BSP using three supersteps. Our prototype implementation proves practical advantages of our approach.

I. Introduction

Research and development of parallelization has 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 structure. XML has achieved the status of the de facto standard for representing structured information. Depending on the information an XML tree represents, the shape of the tree may be imbalanced and its size can be quite huge. This situation calls for an efficient mechanism for treating such data structures in parallel to reduce time and mitigate space capacity.

Recalling the research on parallel treatments on trees, *parallel tree contractions*, first proposed by Miller and Reif [1], 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, parallel tree contraction algorithms assume that the tree structures are kept as linked structures. This assumption has two deficiencies. (1) It is conceivable that trees to perform computation on are given from the outside, especially in the form of serialized (unparsed, or streamed) strings; we need to reconstruct its internal representation, split the result and distribute the fragments among processors. (2) 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 technical report summarizes our new approach for fast parallel tree reduction computations on distributed memory environments. One of its notable characteristics is its data representation. It employs the one dimensional arrays produced by tree traversals over the concerned trees. This representation eliminates troubles explained above: we do not need to reconstruct tree structures from the serialized XML representation (this representation is what we exactly apply computations on!); instead of linked structures we uses arrays, which can fully benefit from caching mechanism; and we can partition and distribute the arrays among processors at our will, making data distribution trouble-free.

We present in this technical report two algorithms for applying computation over tree structures. For simplicity we concentrate ourselves in the computational form named *tree homomorphism* which is a natural definition of computation over trees of unbound degrees (like XML), and imposes a weaker condition than what efficient parallel tree contractions requires. The first algorithm is given in a simple and straightforward way: after distributing subarrays among processors, it evaluates the tree computation in a bottom-up manner based on the idea of *list homomorphism* [2]–[4].

This weaker restriction in tree homomorphism, however, incurs drawbacks in execution: parallel computation complexity depends on the factor of the tree depth. Bringing in a condition equivalent for efficient parallel tree contractions to tree homomorphism, we then perform an improvement over the algorithm by exploiting more parallelizable parts. Further reduction enabled by this additional condition produces a binary tree of size $2\mathrm{P} - 3$ using $\mathrm{O}(\mathrm{N}/\mathrm{P})$ communication and computation costs, where P denotes the number of processors and N the problem size (namely the size of the input tree). The information on the resulting tree structure is obtained by the cost $\mathrm{O}(\mathrm{P})$. Therefore our refined algorithm enjoys not only the benefits of data representation, but also theoretical low complexity: it is implemented under BSP model using three supersteps.

The contributions of our work are briefly summarized as follows:

- *New Viewpoint at Parallel Tree Contractions—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. Tree Computation over its serialized representation much resembles *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 [5] or BSP [6], both of which require $\mathrm{O}(\mathrm{N}/\mathrm{P}\log\mathrm{P})$ execution time. The developed algorithms resembles *parallel parentheses matching* [7], [8] in the point of the data representations, and is implemented efficiently under BSP model.

The technical report is organized as follows. After introducing basic preliminary notations in Sect. II, Sect. III introduces tree homomorphism and the properties of tree traversals. Our first algorithm to evaluate trees in parallel on the serialized representation is shown in Sect. IV. Here, one normal form is also presented. The following Sect. V develops the second algorithm. We show that an additional condition is equivalent to that of parallel tree contractions, and that this condition enables further computation which derives a binary tree of size $2\mathrm{P} - 3$. In Sect. VI we show some experiments to prove the effectiveness

our framework. We discuss the related work, as well as manipulability of our representation, in Sect. VII. Finally we conclude this technical report in Sect. VIII with mentioning future directions.

II. Preliminaries

This section introduces some notational conventions and assumption used in this technical report. We borrow the ideas from functional programming [9] and notations from Haskell [10] for simplicity and preciseness of discussion. Readers may find figures helpful to get the intuition enough to understand the essence of this paper.

A. Constants and model for parallel computation

We assume distributed memory environments. 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 processors. 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 in Sect. V involves all-to-all transactions; such a mechanism is available in MPI on many PC clusters. We assume BSP model [11], [12] when we develop the refined algorithm in Sect. V.

B. Functions and lists

Function application is denoted by a space and the argument which may be written without brackets. Thus $f\ a$ means $f(a)$. Functions are curried, and application associates to the left. Thus $f\ a\ b$ means $(f\ a)\ b$. Function application binds stronger than any other operator, so $f\ a \oplus b$ means $(f\ a) \oplus b$, not $f\ (a \oplus b)$. A centralized circle $\circ$ denotes *Function composition*. By definition, we have $(f \circ g)\ a = f\ (g\ a)$. Function composition is an associative operator, with the identity function id as its unit. Infix binary operators will often be denoted by $\oplus$ or $\otimes$, and can be *sectioned*: an infix binary operator like $\oplus$ can be turned into unary or binary functions by $a \oplus b = (a \oplus)\ b = (\oplus b)\ a = (\oplus)\ a\ b$. We often use *anonymous functions*, functions without their name. They are denoted with lambda expression; for example, $\lambda x.x$ is a function which takes a value and returns it without any change, namely an identity function.

Pairs (or *tuples*) are Cartesian products of plural data, written like (x, y) . For a pair $p = (x, y)$, fst $p = x$ and snd $p = y$ holds. *Lists* (formally *join-* or *append-lists*) are finite sequences of values of the same type. A list is either the empty list, a singleton, or concatenation of two other lists. We write $[\,]$ for the empty list, $[a]$ for the singleton list with element a, and $x{+\!\!+}y$ for the concatenation (join) of two lists x and y. Concatenation is associative, having the unit $[\,]$. For example, $[1]{+\!\!+}[2]{+\!\!+}[3]$ denotes a list with three elements, abbreviated to $[1, 2, 3]$.

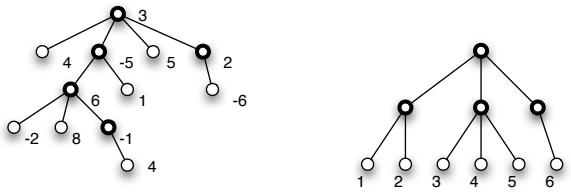
C. List homomorphism

List homomorphism is a model that plays an important role for developing efficient parallel programs [2]–[4], [13]. A function h^L is a homomorphism if there exist an associative operator $\oplus$ and a unary function g such that

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

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 $h^L\ x$ and $h^L\ y$ 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 $\mathit{sum}\ (x{+\!\!+}y) = \mathit{sum}\ x + \mathit{sum}\ y$ and $\mathit{sum}\ [a] = a$ 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



(a) A rose tree (b) Rose tree representation of $[[1,2],[3,4,5],[6]]$

Fig. 1. Rose trees (indicating internal nodes by thick circles, leaves by thin circles)

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 $\mathrm{O}(\mathrm{N}/\mathrm{P} + \log \mathrm{P})$.

III. Trees and Tree Homomorphism

Before analyzing parallelization, this section observes the underlying frameworks, namely the tree structures, its serialized form by tree traversals, and the tree homomorphism.

A. Trees and their list representation

Nested structures like trees or nested lists can take various forms. In order to capture these varieties, we treat a data structure called *rose tree*, which denotes a tree with unbound degree (tree whose node can have an arbitrary number of subtrees).

Definition 1 *A data structure RTree is called* rose tree *defined as follows.*

$$\textit{RTree}\ \alpha = \textit{Node}\ \alpha\ [\textit{Rtree}\ \alpha]$$

□

An example is shown in Fig. 1(a). This structure is general enough to treat nested lists. For example, $[[1,2],[3,4,5],[6]]$ is formatted into a tree shown in Fig. 1(b) where internal nodes do not have their value. XML trees are naturally represented under rose trees; binary trees are also covered by limiting degrees of each node to be either zero or two.

As was explained in Introduction, tree-structured data is held 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 this paper we treat only *well-formed* list representation, which is guaranteed to be parsed into trees. To make such list representation easy to observe, we assume

- to ignore information in the closing tag, and
- 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 tag. The associated depth information can be easily obtained by prefix sum computation [14].

The top sequence in Fig. 2 is the translated list from the example tree of Fig. 1(a). This list is visually aligned by depth as the lower part in Fig. 2. We later see the information of depth helps us to have insight for deriving 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.

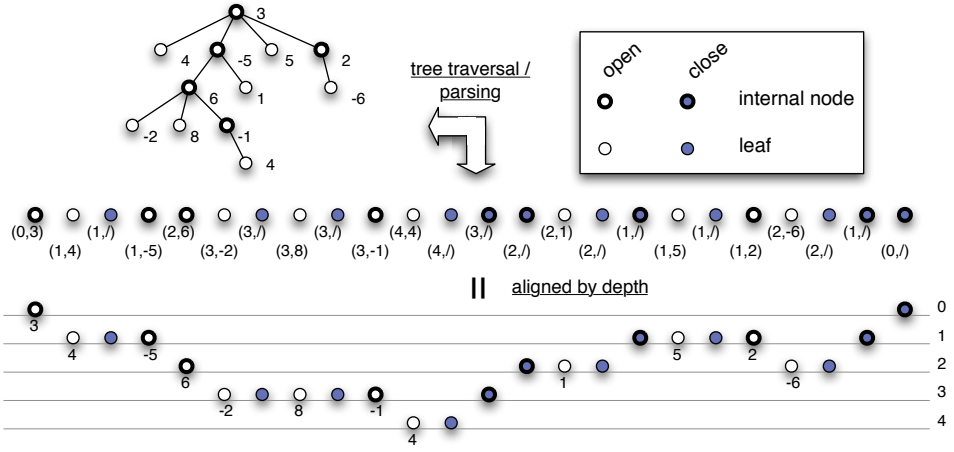


Fig. 2. The list representation of the example rose tree of Fig. 1(a) (the upper as a list of pairs, the lower as a list aligned by the depth)

We confirm a property of these translated lists before developing our algorithm.

Lemma 1 *A sequence of elements starting from a open element at depth d to the first close element at depth d forms a subtree. Similarly, a sequence of elements starting from a open element at depth d to some close element at depth d, before encountering any element at depth smaller than d, forms a sequence of subtrees (namely a* forest*).*

Proof: This lemma is naturally derived from the algorithm of tree traversals. Assume a tree is given whose root has its depth d. The left-to-right tree traversal (1) initially produces a open node of depth d whose value is what the root has, and (2) recursively applies the algorithm on its subtrees by passing the depth $d+1$, and (3) finishes its procedure by producing a close node of depth d. Since (2) guarantees that the result by subtrees always has depth more than d, the first close node of depth d after the open node of depth d by (1) is what (3) produces, which constructs a tree. The repetition in (2) returns a sequence of results the subtrees return, which for each are enclosed by a open and close node of depth $d+1$ without any occurrence of nodes of depth d or smaller. There is no other procedures to traverse, and this fact proves the correctness of this lemma. ■

B. Tree homomorphism

Construction of algorithms goes along with the concerned data structure. A natural general recursive form arises, which we call *tree homomorphism* [15].

Definition 2 *A function h^T defined as follows over rose trees is called* tree homomorphism.

$$h^T\ (\mathit{Node}\ a\ [t_1, \ldots, t_n]) = a \oplus (h^T\ t_1 \otimes \cdots \otimes h^T\ t_n)$$

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

This definition consists of two kinds of computation: among siblings by an operator $\otimes$ and between a parent and its children by $\oplus$. For later convenience, we define $\ominus$ as $(a, b, c) \ominus e = a \oplus (b \otimes e \otimes c)$.

As a simple example, take a tree of unbound degree (e.g. Fig. 1(a)) which has integer weights in its nodes. We want to compute the maximum of the values each of which is a summation of weights of the nodes from the root to each leaf. For the tree in the figure, the result should be 12 contributed by the path $[3, -5, 6, 8]$ from the root. We may define the following homomorphism *maxPath*:

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

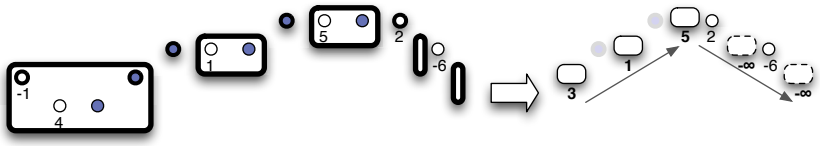


Fig. 3. Reducing a sublist of Fig. 1(a) into a h-NF (indicating computed results by ovals and inserted $\iota_\otimes$ by dashed ovals)

where $\uparrow$ is a binary operator returning the bigger of two numbers. Therefore two parameterized operators $\otimes$ and $\oplus$ are basically instantiated to $\uparrow$ and to $+$, respectively.[1] For other examples please see [16].

For the time being, the condition is just the associativity in the operator $\otimes$. This assumption is weaker than what parallel tree contractions require, since no restriction is imposed to the vertical direction between $\oplus$ and $\otimes$. We introduce an restriction on the relation between them in Sect. V, which are found to be equivalent to that of the parallel tree contractions.

IV. Tree Computation over their List Representation

The first approach is to employ a list homomorphic approach toward tree homomorphism. The task is therefore to figure out what are its corresponding $\odot$ and g, by observing the equivalent computation under list homomorphism. The benefit of list homomorphism lies in the associativity of *list join*: we have freedom in choosing how to split a given list into pieces, in other words, in which order to merge the partial results of these fragments into one final result.

The common cases in list homomorphism are that the partial as well as the final results are just single values produced by the associative operator $\odot$. When the list representations of trees are concerned, however, their sublists do not necessarily form a subtree, in which case tree homomorphism cannot reduce the fragments into a single value. The safe decision is to employ *suspension*: applying computation where it can take place, and leaving other parts as they are.

A. Hill normal form

To illustrate the idea, let us focus on a sublist of Fig. 2 starting from the 10th open element $(3, -1)$ to the 21st open element $(2, -6)$. According to Lemma 1, the four elements starting from the open $(3, -1)$ and two elements starting from the open $(2, 1)$ and $(1, 5)$ form for each a subtree. Recursive definition and the associativity of the operator $\otimes$ guarantees us to evaluate a subtree as well as a forest. We can regard the sequences of zero elements as forests, too, and the value they produce is treated as $\iota_\otimes$. These artificially added values $\iota_\otimes$ appear in the following three cases:

- at depth $d + 1$ of the leftmost position when the first element of the list is close at depth d;
- at depth d between the ascending and descending sequences when there do not appear plural open or close elements at the highest depth d; and
- at depth $d + 1$ of the rightmost position when the last element of the list is open at depth d.

After computing available parts of the given sublist there remains a sequence of list elements and computed values. This process toward the example sublist is shown in Fig. 3, visually aligned by depth.

The bottom of Fig. 3 tells us that the resulting sequence has two parts: an ascending sequence of close elements and computed values alternately on the left; the other sequence of open elements and computed values alternately on the right. We name such a sequence a *hill normal form* (*h-NF* for short) from the visual shape. Note that closes without any value are safely taken out thanks to the existence of computed values or inserted $\iota_\otimes$ appearing between them. Therefore a h-NF is specified by a pair of one value and three lists (d, cs, as, bs), namely the *peak depth*, resulting values of the left, values of opens on the right

[1] In case of signed integers the units of two operators are different ($-\infty$ and 0 for $\uparrow$ and $+$, respectively). We need to modify $+$ to $+'$, where $x +' y$ is defined as "if $y = -\infty$ then x else $x + y$."

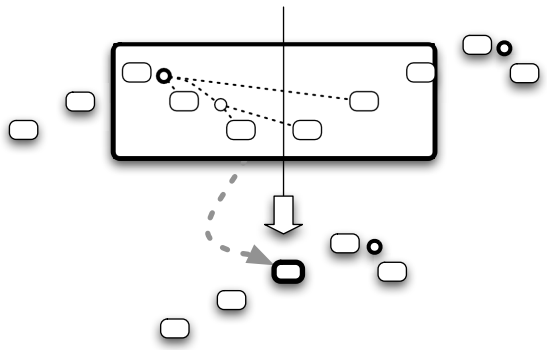


Fig. 4. Merging two adjacent h-NFs (dotted lines indicate links of the original tree)

and resulting values of the right, respectively. Elements in these lists are kept in the order of ascending depth.

Observation 1 *A h-NF has the following properties.*

- as *and* bs *have the same length, and elements of* bs *have depth larger by one than the corresponding elements of* as*.*
- *The peak depth* d *denotes the highest point of the hill.* cs *and* as*, if not empty, share the same peak depth.* □

The example sublist is reduced into a h-NF (d, cs, as, bs) of

$$\begin{aligned} d &= 1, \\ cs &= [-1 \oplus (4 \oplus \iota_\otimes), (1 \oplus \iota_\otimes), (5 \oplus \iota_\otimes)] \ , \\ as &= [-6, 2] \ , \\ bs &= [\iota_\otimes, \iota_\otimes] \ . \end{aligned}$$

To formalize the behavior, we see what happens when a singleton list is given.

Routine 1 *If the element in the singleton list is open* (d, v)*, then the h-NF is*

$$(d,\ [\iota_\otimes],\ [v],\ [\iota_\otimes]) \ ;$$

otherwise (namely close $(d, /)$*)*

$$(d,\ [\iota_\otimes, \iota_\otimes],\ [\,],\ [\,]) \ . \qquad \square$$

When we partition the list representation in Fig. 2 evenly in four processors, the resulting shape of h-NFs will be like in Fig. 5.

B. Merging hill normal forms

Next we clarify $\odot$ by considering the case a join of two sublists $z = x +\!\!+ y$ is given. The joined sublist z is again a sublist of the list representation, and it is to form a h-NF. The question is how we obtain, from two h-NFs $x' = (d_x, cs_x, as_x, bs_x)$ and $y' = (d_y, cs_y, as_y, bs_y)$, coming from x and y, respectively, a new value. This calculated value z' is to represent the result corresponding to z, hence it is again a h-NF $z' = (d_x, cs_z, as_z, bs_z)$. Here Lemma 1 tells us that a sequence starting from the open element of x' to the (already erased) close element of y' of the same depth forms a subtree, and evaluating the longest of such sequences produces again a h-NF (see Fig. 4). This is exactly what we would have when we are given the sublist z and to compute its corresponding h-NF. Due to the rule to produce a h-NF from a sublist explained up above, the deepest point of bs_x and cs_y share the same depth, by observing open/close of the rightmost of x and the leftmost of y.

The routine for merging two h-NFs are formalized as follows.

Routine 2 *Assume sublists x and y have h-NFs $x' = (d_x, cs_x, as_x, bs_x)$ and $y' = (d_y, cs_y, as_y, bs_y)$, respectively. Index accesses are assumed on lists, where indexes start from 1. With $|x|$ denoting the length of a list x, the h-NF $z' = (d_x, cs_z, as_z, bs_z)$ of the list join $z = x +\!\!+ y$ is computed as follows.*

If $d_x < d_y$, with $n \leftarrow |c_y|$ and $m \leftarrow |a_y|$:

$$
\begin{array}{lcll}
a_{z\ [i]} & \leftarrow & a_{y\ [i]} & (1 \le i \le n) \\
a_{z\ [m+i]} & \leftarrow & a_{x\ [n-1+i]} & (1 \le i \le |a_x| - n + 1) \\
b_{z\ [i]} & \leftarrow & b_{y\ [i]} & (1 \le i \le n) \\
b_{z\ [m+1]} & \leftarrow & a_{x\ [n]} \otimes \mathit{filled\_valley} \otimes c_{y\ [n]} & \\
b_{z\ [m+1+i]} & \leftarrow & b_{x\ [n+i]} & (1 \le i \le |a_x| - n) \\
c_{z\ [i]} & \leftarrow & c_{x\ [i]} & (1 \le i \le |c_x|)\ ,
\end{array}
$$

otherwise (namely $d_x \ge d_y$), with $n \leftarrow |a_x| + 1$ and $m \leftarrow |c_x|$:

$$
\begin{array}{lcll}
a_{z\ [i]} & \leftarrow & a_{y\ [i]} & (1 \le i \le |a_y|) \\
b_{z\ [i]} & \leftarrow & b_{y\ [i]} & (1 \le i \le |b_y|) \\
c_{z\ [i]} & \leftarrow & c_{x\ [i]} & (1 \le i \le m-1) \\
c_{z\ [m]} & \leftarrow & c_{x\ [\,|c_x|\,]} \otimes \mathit{filled\_valley} \otimes c_{y\ [n]} & \\
c_{z\ [m+i]} & \leftarrow & c_{y\ [n+i]} & (1 \le i \le |c_y| - n)\ ,
\end{array}
$$

where the value of $filled\_valley$ is obtained by:

$$
\begin{array}{l}
\mathit{fill\_valley}\ \{ \\
\quad v \leftarrow \iota_{\otimes} \\
\quad \textit{for}\ i \leftarrow 1\ \ldots\ n{-}1 \\
\qquad \{v \leftarrow a_{x\ [i]} \oplus (b_{x\ [i]} \otimes v \otimes c_{y\ [i]})\} \\
\quad \textit{return}\ v \\
\}\ .
\end{array}
$$

□

When all of recursive list joining finish, just a single value remains in cs and this is the desired result of the tree homomorphism. It is easily verified that the merging operation is associative, since merging operation is to reduce cs of the left h-NF and as and bs of the right, and the operator $\otimes$ is associative.

C. The first algorithm and its complexity

As the summary of this subsection, we show our algorithm which implements tree homomorphism as list homomorphism.

Algorithm 1 *Tree homomorphism in Definition 2 is reformatted as list homomorphism, where*

- *g is defined as Routine 1, and*
- *$\odot$ is defined as Routine 2.*

□

Now we estimate the parallel complexity of this algorithm. Assume all processors have the same number of list elements (namely $2\mathrm{N}/\mathrm{P}$). Algorithms specified as list homomorphism has two phases: (a) the local computation at each processor, and (b) the global computation between these processors. Each processor produces its corresponding h-NF in Phase (a). It requires linear cost to the length of the list, namely $\mathrm{O}(\mathrm{N}/\mathrm{P})$. With associative operator $\odot$, the computational cost of Phase (b) is $c_{\odot} \log \mathrm{P}$, where $c_{\odot}$ is the cost to compute $\odot$. This cost depends on the sizes of h-NFs, which are at most the depth of the tree D. The cost for data transaction in each step of Phase (b) is linear to the size of h-NFs in addition to

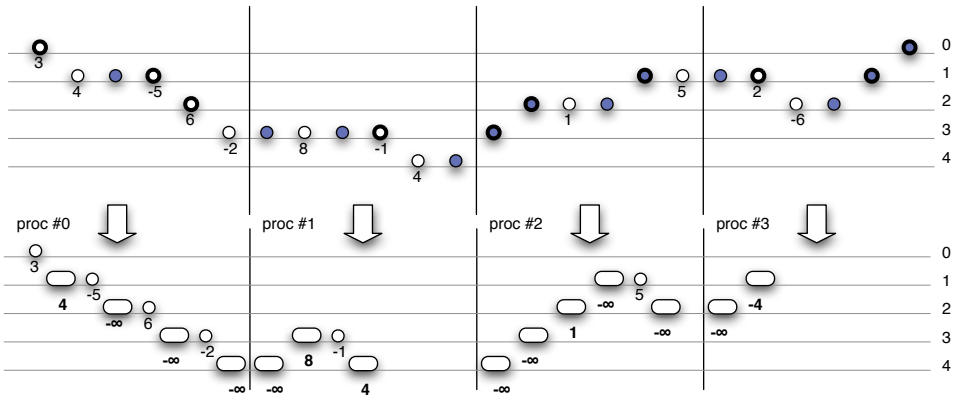


Fig. 5. Four h-NFs from the list representation in Fig. 2

the cost of communication start-up. This cost is also affected by the depth D. Therefore overall cost is $\mathrm{O}(\mathrm{N}/\mathrm{P} + \mathrm{D}\log \mathrm{P})$.

Note that the peak depth and the depths at processor boundaries matter to the cost of computation and transaction. From the analysis on the *path length* (the sum of each node's depth) [17], the average depth $\mathrm{O}(\sqrt{\mathrm{N}})$ may give us some hint. In case of rose trees, especially XML, their depth can be much shallower. The worst case is when monadic trees (trees of nodes whose degree is at most 1) is given. The height of h-NFs doubles every iteration in Phase (b) until N, and at last one processor applies computation to the whole representation. The complexity therefore results in $\mathrm{O}(\mathrm{N})$.

V. Refined Algorithm using Conditions for Parallel Tree Contractions

The h-NFs and the merging computation presented in the previous section were straightforwardly defined. The simplicity of list homomorphism makes the algorithm development easy, at the expense of unnecessary data transactions as h-NFs from one processor to another without any computation. It is acceptable when we know the structures in concern are shallow enough; partitioning of the list representation in any way creates h-NFs of small size. But this is not always the case. As is guaranteed under parallel tree contractions, the desired algorithm needs to be freed from the concerns on the shape of the tree and the depth factors.

In this section we apply refinement and develop a new efficient algorithm. It employs an additional condition in tree homomorphism, and as a result reduces the tree computation into tree contractions over a binary tree of size $2\mathrm{P} - 3$. We explain the idea of our algorithm using the running example in Figs. 1(a) and 2.

A. Tree homomorphism and conditions for parallel tree contractions

Tree homomorphism is defined in a bottom-up manner, and this characteristics requires whole subtrees to be evaluated before evaluating a node. While the condition of tree homomorphism is flexible for users, it is restrictive for efficient parallel execution. We employ an additional condition on operators of tree homomorphisms, and we show it is equivalent to the conditions for parallel tree contractions.

Fig. 5 shows the resulting sequence of h-NFs when the list reporesentation is evenly distributed among four processors (For convenience of discussion, we omit the value in cs_0 and $cs_{\mathrm{P}-1}$ at depth 0 since they are always $\iota_{\otimes}$). When we look carefully at the figure, 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 $-\infty$ in cs_3 (this correspondence might be clearer in Fig. 8). 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

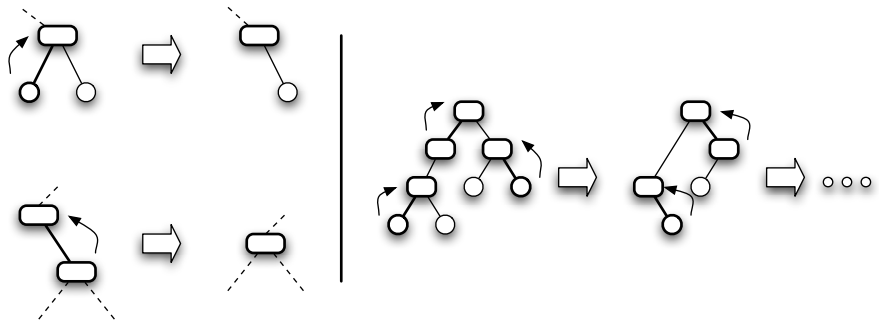


Fig. 6. Tree contractions *rake* (upper left) and *compress* (upper left), and parallel tree contractions (right)

time being). We notice that the value 3 in as_0 with these two values in processors 0 and 3 forms a *triple* $(3, 4, -\infty)$.

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 $(6, -\infty, -\infty)$. Using the operator $\ominus$ defined in Sect. III the computation implied by these is

$$(\ ((-5, -\infty, 1)\ominus) \circ ((6, -\infty, -\infty)\ominus)\ )\ e\ ,$$

and awaits values from subtrees whose value is e. To compute this sequence of composition beforehand, a sufficient condition is that the fragments $((a, b, c)\ominus)$ are closed under function composition: given two fragments $((a_u, b_u, c_u)\ominus)$ and $((a_l, b_l, c_l)\ominus)$, there exist three values (a', b', c') such that for any x

$$(a_u, b_u, c_u) \ominus ((a_l, b_l, c_l) \ominus x) = (a', b', c') \ominus x\ .$$

Such property is known as *context preservation* [18].

This condition has already been formalized as *extended distributivity* [16]. There, $((a, b, c)\ominus)$ was named as *distributive normal form* (*d-NF* in this paper) and the corresponding p_a, p_b, p_c as *characteristic functions* of distributive normal forms.

Definition 3 *The operator $\otimes$ is said* extended distributive *over $\oplus$ when sectioned functions $((a, b, c)\ominus) = \lambda x.a \oplus (b \otimes x \otimes c)$ are closed under function composition, namely*

$$(\lambda x.a_u \oplus (b_u \otimes x \otimes c_u)) \circ (\lambda y.a_l \oplus (b_l \otimes y \otimes c_l)) = \lambda y.a' \oplus (b' \otimes y \otimes c')$$

when we can find appropriate functions p_a, p_b and p_c which calculate

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

□

For example, take the example of *maxPath* in Sect. III-B. The operators $\uparrow$ and $+$ satisfy closure property:

$$\begin{array}{ll} & (\lambda x.a_u + (b_u \uparrow x \uparrow c_u)) \circ (\lambda y.a_l + (b_l \uparrow y \uparrow c_l)) \\ = & \lambda y.(a_u + b_u) \uparrow (a_u + a_l + b_l) \uparrow (a_u + a_l + y) \uparrow (a_u + a_l + c_l) \uparrow (a_u + c_u) \\ = & \lambda y.(a_u + a_l) + ((b_u - a_l \uparrow b_l) \uparrow y \uparrow (c_l \uparrow c_u - a_l)) \\ = & \lambda y.a' + (b' \uparrow y \uparrow c')\ ; \end{array}$$

where the characteristic functions[2] are implicitly

$$\begin{array}{rcl} a' & = & a_u + a_l\ , \\ b' & = & b_u - a_l \uparrow b_l\ ,\ \text{and} \\ c' & = & c_l \uparrow c_u - a_l\ . \end{array}$$

[2]These characteristic functions also hold when $\oplus$ is $+'$ of the footnote in Sect. III-B.

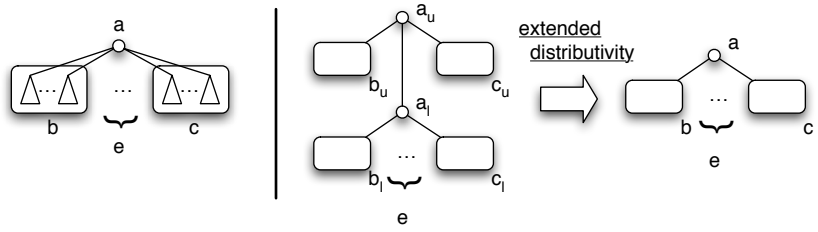


Fig. 7. A *triple* (left) and extended distributivity (right)

Other examples can be found in [16].

Now we show this property is equivalent to the condition for parallel tree contractions 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. 6).

Theorem 1 *Assume computation of $\oplus$ and $\otimes$ requires constant time. The closure property of d-NFs is equivalent to the conditions for parallel tree contractions hold 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 $((a, \iota_\otimes, \iota_\otimes)\ominus) = \lambda x.a \oplus (\iota_\otimes \otimes x \otimes \iota_\otimes)$. Take its leftmost leaf whose computed value is b. It has its siblings whose value become y, and this can be written as $\lambda y.b' \otimes y$. The rake operation is to merge $\lambda y.b' \otimes y$ into $\lambda x.a \oplus (\iota_\otimes \otimes x \otimes \iota_\otimes)$, namely

$$\begin{aligned}(\lambda x.a \oplus (\iota_\otimes \otimes x \otimes \iota_\otimes) \circ (\lambda y.b' \otimes y) &= (\lambda y.a \oplus (\iota_\otimes \otimes (b' \otimes y) \otimes \iota_\otimes) \\ &= \lambda y.a \oplus (b' \otimes y \otimes \iota_\otimes)\end{aligned}$$

holds and successfully the leaf of value b is raked. Similarly for the value c' of its rightmost leaf, the rake operation produces $\lambda y.a \oplus (\iota_\otimes \otimes y \otimes c')$. The same arguments apply to the nodes which already have raked values, namely $\lambda x.a \oplus (b \otimes x \otimes c)$.

The compress operation is written as

$$(\lambda x.a_u \oplus (b_u \otimes x \otimes c_u)) \circ (\lambda x.a_l \oplus (b_l \otimes x \otimes c_l)) \ .$$

The assumption of closure property reduces it into $(\lambda x.a' \oplus (b' \otimes x \otimes c'))$ using its characteristic functions p_1, p_2 and p_3 . ■

The value parts of d-NFs, namely (a, b, c), are initially considered to denote a tree whose root has a value a and subtrees are consecutively missing; its existing subtrees from the left and the right are reduced by tree homomorphism into single values b and c, respectively (see Fig. 7, left). These *triple*s are gradually merged by this closure property of d-NFs which is rephrased as: "Given a d-NF, and its missing part is one subtree specified as another d-NF, then these two can be merged into one." (See Fig. 7, right.) Note that the unit element of d-NFs is $((\iota_\oplus, \iota_\otimes, \iota_\otimes)\ominus)$.

Note that sequences of d-NFs coming from two adjacent processors are reduced into a value without any missing subtrees in between: instead of treating using extended distributivity which is often more expensive, we apply ordinary reduction computation to obtaion the computed values $-2 \oplus \iota_\otimes = -2$, $-1 \oplus (4 \otimes \iota_\otimes) = 3$, $5 \oplus (\iota_\otimes \otimes -4) = 1$ from 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} ($i = 0, 1, 2$), respectively.

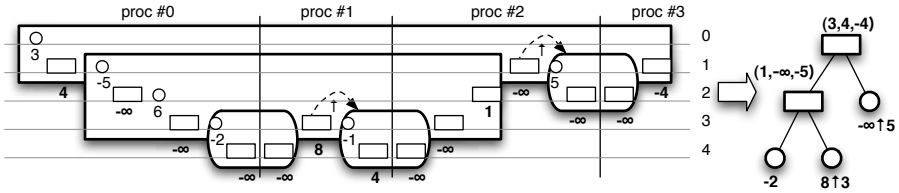


Fig. 8. A binary tree of the running example after the mating process

B. Reformatting into a binary tree

By introducing the additional condition, the enclosed part is reduced to a single d-NF $((a', b', c')\ominus)$. Similarly, other pairs of facing slopes consisting of as, bs from the left and cs from the right are reduced into either single d-NFs or some single values. We here investigate such *mating pairs.*

We first observe how mating pairs are distinguished. The h-NF in each processor has its peak depth. It is easily seen that the h-NF in the first processor has its peak depth 0, and the h-NF of the last processor $\mathrm{P}-1$ as well. If there are any h-NFs in between, then we can always find one h-NF (or more) whose peak depth d is smaller than any other next to the both ends (processor 2 corresponds to it in Fig. 4). When there is just one such peak in one h-NF, it divides the the structure into three: a pair (in the dashed part) and the left and right subsequences of h-NFs. The both ends of each obtained subsequence are regarded to have peak depth d, and the mating process continues recursively.

The following lemma estimates the number of mating pairs.

Theorem 2 *We denote the number of mating pairs of facing slopes as R_{P}. Then $R_{\mathrm{P}} \leq 2\mathrm{P} - 3$.*

Proof: For simplicity we assume the peak depths of h-NFs are disjoint for each other except for the first and last ones; under this assumption the equality $R_{\mathrm{P}} = 2\mathrm{P} - 3$ holds. Proof is given by mathematical induction.

Consider the case $\mathrm{P} = 2$. There is no other h-NFs in between and we know $R_2 = 1 = 2 \cdot 2 - 3$. Assume $R_i = 2i - 3$ holds for $i \leq p - 1$, and a sequence of p h-NFs is given. By the assumption of disjoint peak depths there exists just one h-NF which has the smallest peak depth next to the both ends, and divides a sequence of h-NFs into three parts as mentioned above. If one subsequence has its length j the length of the other is $p - j + 1$. Hence with induction hypothesis

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

holds, which finishes the proof.

Inequality appears in case there are h-NFs of the same peak depth appearing adjacently. For example, assume there are four processors and h-NFs in processors 1 and 2 have the same peak depth, namely $d_1 = d_2 > d_0 = d_3$. In this case, the mating pairs are created between processors 0 and 3, 0 and 1, 1 and 2, 2 and 3. This partitioning produces one smaller value than $2 \cdot 4 - 3 = 5$. Generalization of this argument proves $R_{\mathrm{P}} \leq 2\mathrm{P} - 3$. ■

To avoid inequality in R_{P} we add dummy mating pair which corresponds to the unit of d-NFs.

The following procedure returns information on mating pairs in linear time to the number of h-NFs (namely P). Here a mating pair is written as $M^{i \leftrightarrow j}_{[u,l]}$. This implies a mating pair between left processor i and right processor j, of depths from u to l.

Routine 3 *Assume a list of pairs (p, d_p) is given in the ascending order of p, where p and d_p denotes the processor number $0, \ldots, \mathrm{P} - 1$ and the peak depth of each h-NF, respectively.*

Push the first pair $(0, 0)$ on a stack; the pair at the top of the stack is referred as (p_s, d_s) .

For each i of $\{1, \ldots, \mathrm{P}-1\}$:

- *Prepare a variable $d \leftarrow \infty$*
- *While $d_i < d_s$,*
  - *produce $M^{p_s \leftrightarrow i}_{[d_s,d]}$,*
  - *$d \leftarrow d_s$, and*
  - *pop from the stack.*
- *If $d_i = d_s$, then*
  - *produce $M^{p_s \leftrightarrow i}_{[d_s,d]}$, then $M^{\leftrightarrow}_{[d_i,d_s]}$, and* — *(*)*
  - *pop from the stack;*

  else
  - *produce $M^{p_s \leftrightarrow i}_{[d_i,d]}$.*
- *Push (i, d_i).*

Finally eliminate the last mating pair (that is $M^{\leftrightarrow}_{[0,0]}$). □

The first mating pair produced in each iteration, which is always in the form $M^{i \leftrightarrow i+1}_{[u,\infty]}$, denotes a single value. By ∞ the whole list elements starting up to the depth u or $u+1$ are mated. The other pairs imply d-NFs, and the pair produced at (*) is the dummy mating $M^{\leftrightarrow}_{[d,d]}$.

It should be noted that these mating pairs construct a binary tree (consisting of $2\mathrm{P}-3$ elements), and that the order of occurrence is *postorder*. The corresponding binary tree for our running example is shown in Fig. 8. Since internal nodes are d-NFs, tree contractions over this binary tree are efficiently executed.

We need pay attention to the elements of cs at its peak depth. These values spilt out of the mating pairs when they are in processors $i, \ldots \mathrm{P}-2$. We associate such values in cs to the mating pair on its right. In our example of Fig. 8, the value 8 of depth 3 in processor 1 and the value $-\infty$ of depth 1 in processor 2 are associated to $M^{1 \leftrightarrow 2}_{[3,\infty]}$ and $M^{2 \leftrightarrow 3}_{[1,\infty]}$, respectively.

C. Processor allocation

The remaining question is to decide allocation of mating pairs. The number of mating pairs is $2\mathrm{P}-3$, and the binary tree is to have $\mathrm{P}-1$ leaves. While any deterministic procedure works well, one idea is that one processor is assigned to two different kinds of nodes, namely one leaf and one internal node.

Routine 4 *Assume the sequence of mating pairs which is the result of Routine 3 is given. We use a stack, and three variables v_1, v_2, and v_3.*

For each mating pair $M^{i \leftrightarrow j}_{[u,l]}$ in the resulting order, except for the last one:

- *If it is a leaf $M^{i \leftrightarrow i+1}_{[u,l]}$, then*
  - *assign processor $i+1$ to the mating, and*
  - *$v_3 \leftarrow (\texttt{leaf}, i+1)$.*

  else
  - *pop from the stack to $v_1 \leftarrow (t_1, p_1)$*
  - *pop from the stack to $v_2 \leftarrow (t_2, p_2)$*
  - *if t_1 is* `leaf` *then assign p_1 to the mating, and $v_3 \leftarrow v_2$;*
    else if t_2 is `leaf` *then assign p_2 to the mating, and $v_3 \leftarrow v_1$;*
    else assign p_1 to the mating, and $v_3 \leftarrow (\texttt{node}, p_2)$.
- *Push v_3 to the stack.*

Processor 0 is assigned to the last mating. □

An assignment example is depicted in Fig. 9 using a binary tree of 13 nodes (it occurs when $\mathrm{P} = 8$). We set to rewrite a mating pair $M^{i \leftrightarrow j}_{[u,l]}$ assigned to processor p as follows: $M^{i \rightarrow j}_{[u,l]}$ (when $p = j$), $M^{i \leftarrow j}_{[u,l]}$ (when $p = i$), and $M^{i \rightarrow p \leftarrow j}_{[u,l]}$ (otherwise).

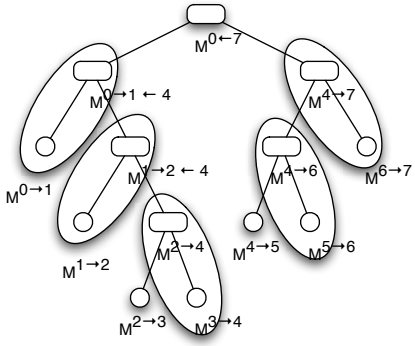


Fig. 9. Processor allocation for a binary tree of 13 nodes

This processor assignment is efficient in terms of data transaction in the following two aspects. (1) Processors already have one fragment data of mating pairs in most cases. Take the leaf pair $M_{[u,l]}^{i\to i+1}$ for example. This pair consists of a part of as and bs in the processor i, and a part of cs in the processor $i+1$. The processor $i+1$ is to reduce it into some value, and the processor can evaluate this mating by receiving information from the processor i . The similar behavior applies to internal nodes having a leaf on their right, and indeed Routine 4 tries to assign in this way. (2) In case an internal node has one or more leaves, one processor takes care of both of the internal node and one of its direct leaves. Routine 4 again tries to assign the same processor to as many connected two nodes as possible. Due to the additional condition employed in Sect. V-A, these two nodes are merged into one single d-NF without any transactions among processors.

D. The refined algorithm and its complexity

As the summary of this section, we sum up the procedures into one algorithm. As is mentioned in Sect. II, we assume BSP model here.

Algorithm 2 *Assume the list 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 $- 1$.

- *(a) Each processor sequentially performs tree computation, and creates a h-NF. Along with this computation, each processor finds its peak depth.*
- *(b) All peak depths are shared through global communication using all-to-all transactions.*
- *(c) Each processor computes the mating pairs and the resulting binary tree using Routine 3; allocation of mating pairs and data transaction among processors are also computed using Routine 4.*
- *(d) Transmit fragments of mating pairs to their corresponding processor according to the information obtained in Step (c).*
- *(e) Each processor reduces sequences of mating pairs into single values or single d-NFs.*
- *(f)*
  1) *Single values and d-NFs obtained in each processor are collected in processor 0.*
  2) *Processor 0 reduces the binary tree into the result.*

□

We estimate the cost of this algorithm. We write $\mathbb{M}$ as the set of all mating pairs $M_{[u,l]}^{i\leftrightarrow j}$. Its subsets are denoted as follows: $\mathbb{M}_{i\to}$ denotes the subset of $\mathbb{M}$ whose elements has i as source of data transaction (namely $M_{[u,l]}^{\underline{i}\to j}$, $M_{[u,l]}^{j\leftarrow\underline{i}}$, $M_{[u,l]}^{\underline{i}\to p\leftarrow j}$, or $M_{[u,l]}^{j\to p\leftarrow\underline{i}}$, where underline are given for clarity); $\mathbb{M}_{\to i}$ denotes the subset of $\mathbb{M}$ whose elements has i as destination of data transaction (namely $M_{[u,l]}^{j\to\underline{i}}$, $M_{[u,l]}^{\underline{i}\leftarrow j}$, or $M_{[u,l]}^{j\to\underline{i}\leftarrow p}$ where underline are given for clarity).

TABLE I

COST ESTIMATION OF ALGORITHM 2 UNDER BSP

Step	computation	communication	synch.
(a)	$C_{1,2} \cdot 2\mathrm{N}/\mathrm{P}$		
(b)		$g \cdot \mathrm{P}$	L
(c)	$C_{3,4} \cdot \mathrm{P}$		
(d)		$\max_i\{g \cdot (\#\mathbb{M}_{i\rightarrow} + \Sigma\|m\|)\}$ for $0 \leq i \leq \mathrm{P}-1$, $\forall m \in \mathbb{M}_{i\rightarrow}$	L
(e)	$C_{ed} \cdot \max_i\{\Sigma\|m\|\}$ for $0 \leq i \leq \mathrm{P}-1$, $\forall m \in \mathbb{M}_{\rightarrow i}$		
(f)-1		$\max_i\{g \cdot \#\mathbb{M}_{\rightarrow i}\}$	L
(f)-2	$C_{ed} \cdot (\ (2\mathrm{P}-3)-1\ )$		

$\#\mathbb{M}$ denotes the number of elements in the set $\mathbb{M}$. $|M^{i\leftrightarrow j}_{[u,l]}|$ denotes the length, and $|M^{i\leftrightarrow j}_{[u,l]}| = l - u$ if $l \neq \infty$. $C_{1,2}$ and $C_{3,4}$ are, respectively, the computational costs required for Routines 1 and 2, and Routines 3 and 4. C_{ed} is the computational cost for closure composition by extended distributivity.

The cost is summarized in Table I. Step (a) takes time linear to the data size in each processor, which is the same as Algorithm 1. The memory requirement of this sequential algorithm is $\mathrm{O}(\mathrm{N}/\mathrm{P})$ as well. Step (c) takes $\mathrm{O}(\mathrm{P})$ time and space.

Step (d) requires detailed analysis. The cost depends on how each processor sends out its fragment of data. The exact amount can be known from the set of mating pairs $\mathbb{M}$. Given a processor i, $\mathbb{M}_{i\rightarrow}$ denotes the subset of mating pairs where the processor i is the source to send out its data. The processor i requires communication linear to the number of elements in $\mathbb{M}_{i\rightarrow}$, and communication cost linear to the summation of data fragments of $\mathbb{M}_{i\rightarrow}$ kept in the processor i. When we analyze the worst case, it is possible that a processor communicates to every other processor, and that a processor sends out all of its fragments whose size can be at most $2 \cdot 2\mathrm{N}/\mathrm{P}$ (when all of data reside in the right side of the hill). Indeed some of l in $M^{i\leftrightarrow j}_{[u,l]}$ have $-\infty$, please note that they are just place holders and that the total amount of data which appear after Step (a) is at most $2 \cdot 2\mathrm{N}/\mathrm{P}$. Therefore the worst cost is estimated as $g \cdot (\mathrm{P}-1) + 2 \cdot 2\mathrm{N}/\mathrm{P}$, and this is $\mathrm{O}(\mathrm{N}/\mathrm{P})$ under the assumption $\mathrm{N}/\mathrm{P} > \mathrm{P}$.

Similar analysis applies to Step (e), by changing the viewpoint from the transmitter to the receiver. The computational cost of each processor i is $C_{ed} \cdot \max_i\{\Sigma|m|\}$ for $\forall m \in \mathbb{M}_{\rightarrow i}$, and from Routine 4 that each processor receives at most 2 mating pairs, each of whose length is at most N/P. The resulting complexity is $\mathrm{O}(\mathrm{N}/\mathrm{P})$.

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

We conclude this section by stating the following theorem.

Theorem 3 *Tree homomorphism with extended distributivity has a BSP implementation with three supersteps of at most* $\mathrm{O}(\mathrm{P} + \mathrm{N}/\mathrm{P})$ *communication cost for each.*

What we showed in Steps (d) and (e) is somehow pessimistic, and in reality the computational cost is not that bad, as experiments in the next section demonstrates. In the related work He and Huang analyzes that the communication cost is "bounded above by a p-relation for each communication phase," in the average case of ANSVP (all nearest smaller values problem), generalization of parentheses matching problem. [8]

VI. EXPERIMENTS AND DISCUSSION

To validate our algorithm in the real world, we performed experiments using our prototype implementation. It has been written using C++ and MPI. We simulated a simple query on rose trees where local computation of $\otimes$ and $\oplus$ were matrix multiplication-like operations over matrices of the size 10×10.

TABLE II

EXECUTION TIMES OF OUR EXPERIMENTS

		Deep (RD)		Shallow (RS)		Monadic (M)	
P	dist.	comp.	total	comp.	total	comp.	total
1	—	23.5	23.5	22.8	22.8	N.A.	N.A.
2	0.48	12.3	12.7	12.6	13.0	N.A.	N.A.
4	0.56	6.13	6.70	5.81	6.36	60.1	60.7
8	0.62	3.18	3.80	3.79	4.41	20.9	21.5
16	0.70	1.81	2.51	1.96	2.65	14.8	15.5
32	0.77	1.11	1.88	1.04	1.81	8.36	9.14
64	0.83	0.68	1.52	0.57	1.40	4.47	5.30

Fig. 10. Plots of Table II by total execution time (left) and by speedups of computation time (right)

A. Under a homogeneous environment

The underlying environment was a PC cluster of 64 Xeon 2.8 GHz CPU machines with 2GB memory each that are connected via Gigabit Ethernet; softwares were Linux 2.4.21, gcc 2.96, and mpich 1.2.7.

Three randomly generated trees of size 2,000,000 were used: (RS) a shallow tree with maximum height of 7 (avg. 3.89); (RD) a deep tree with maximum height of 5,000 (avg. 3009); and (M) a monadic tree. A monadic tree is a tree whose internal nodes have exactly one child (i.e. a tree-view of a list). The tree size was constrained by the memory size one machine could hold. The height of RS came from observations on XML documents in general [19]. We executed the program over each tree using 2^i processors ($i = 0, \ldots, 6$).

The results of experiments are summarized in Table II. As their plots in the left of Fig. 10 indicates, our algorithm exhibited good scalability for both the shallow (RS) and the deep (RD) trees. Results of the monadic tree (M) fell behind the other two: it ran out of memory until 2 processors (the result by 4 processors seems also affected), and its execution was around seven times slower than other cases. Our algorithm toward monadic trees fails to apply computation by tree homomorphism and leaves uncomputed values in the first phase, and suffers from heavy data transactions costly computation by extended distributivity during the second phase. It was shown that the algorithm of ANSVP problem does not suffer from such heavy transactions toward random permutations [8]. Although this analysis does not directly hold to our internal representations which are not random permutations, experiments on random trees (RS and RD) suggest that we can expect similar favorable effects. Even if the worst cases of monadic trees are concerned, we observed the potential of scalability. In terms of costs of initial data distribution, it is natural that no difference existed by tree shapes except for negligible errors.

We make a brief comparison with existing approaches of parallel tree computations. One common approach under distributed memory environments has their basis in list ranking, and their parallel tree contractions require $\mathrm{O}(\mathrm{N}/\mathrm{P}\log\mathrm{P})$ [5], [6], [20]. This approach is not cost optimal, and additionally communication among processors appears in each $\mathrm{O}(\log\mathrm{P})$ communication round. It is true we have to pay some price in the worst cases, but our experiments showed such cases stayed exceptional. Our algorithm can be coded as simple for-loops over one-dimensional arrays and benefits compiler optimizations well.

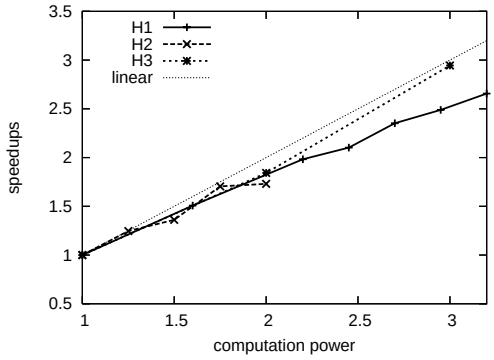


Fig. 11. Experimental results on heterogeneous environments

We also observed that high memory locality by our representations brought considerable performance improvements especially when the involved computation was memory-intensive.

Our existing library implements parallel tree contractions in $\mathrm{O}(\mathrm{N}/\mathrm{P} + \mathrm{P})$ over trees kept as linked structures [16], [21], [22]. This complexity is achieved using m-bridge [23] for initial data distribution and scheduling after the local computation. The theory of m-bridge splits an arbitrary tree (or graph) into smaller fragments of almost the same size. The trouble of this technique is its computation cost. The data distribution process in this paper is really inexpensive, and it was several times faster than that of m-bridge.

B. Under a heterogeneous environment

The second set of experiments is to verify the applicability to heterogeneous environments. For this experiments we used another randomly generated trees: (RS') a shallow tree of 500,000 nodes with maximum height of 7, (RD') a deeper tree of 500,000 nodes with maximum height of 5,000, and (M') a monadic tree of 100,000 nodes. Details of the environment are as follows: the hardware is a PC cluster of (PC-A) four Pentium-4 Xeon 2.0 GHz dual-CPU SMPs with 1GB memory, (PC-B) one Pentium-4 Xeon 1.2 GHz dual-CPU SMP with 1GB memory, (PC-C) and one Pentium-3 Xeon 550 four-CPU SMP with 512MB memory, which are connected with Gigabit Ethernet; and the software is FreeBSD 4.10, gcc 3.4, and mpich 1.2.2. We executed the program on three sets of PC clusters: (H1) a heterogeneous environment with one CPU of PC-A, two CPUs of PC-B, and four CPUs of PC-C, (H2) a heterogeneous environment with one CPU of PC-A and four CPUs of PC-C, (H3) a homogeneous environment with three CPUs of PC-As. Fig. 11 shows the results of the computation times without the initial distribution. The horizontal axis expresses the computation power with respect to that of one CPU of PC-A. Using all the processors in H1, we obtain 2.65 times speedups where the computation power is 3.2 times of one CPU of PC-A. These results show good scalability, and indicate the efficiency also under heterogeneous environments.

VII. Related Work and Discussion

This section compares our proposed framework with related work. It mainly spans parallel tree contractions and list ranking algorithms, flattening transformation, and parentheses matching. In parentheses matching we discuss about the flexibility of the data representation we use in this paper.

A. Parallel tree contractions

Parallel tree contractions, first proposed by Miller and Reif [1], are very important parallel algorithms for trees. Many researchers have devoted themselves to developing efficient implementations on various parallel models [5], [6], [20], [24]–[28]. Among researches based on shared memory environments,

Gibbons and Rytter developed an optimal algorithm on CREW PRAM [24]; Abrahamson et al. developed an optimal and efficient algorithm in $O(N/P + \log P)$ on EREW PRAM [26].

Recently parallel tree contractions as well as list ranking, which serves the basis of 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 [5], [20]; Dehne et al. solved list ranking and tree contractions on CGM/BSP using $O(N/P \log P)$ parallel time [6]. Sibeyn proposed a list ranking algorithm which aimed at reduction of communication costs [29].

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, involves 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 fails to benefit from caching effects.

Especially the advantage applies to the case when trees are given in XML format. To apply related work, we first need to construct a tree from XML form, and apply parallel tree computation over the obtained internal representation. What we have done is the *shortcutting* of these processes. We can also make a note that the XML format is the result of Euler tour, and we take its advantage.

The second advantage is the cost and concerns of data distribution. Under sophisticated distribution of tree structures we can employ the EREW-PRAM tree contraction algorithm. One technique to fulfill this is m-bridges [23], 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 previous work has chosen this approach [21], [30]. The algorithm for m-bridges requires Euler tour and list ranking under distributed memory environments, however, which spoils the theoretical complexity of m-bridges. In this topic we can claim the following two points. First, we are free to split the representations at our will, and the size can easily adjusted. This flexibility is quite beneficial in terms of data distribution among processors. is heterogeneous. Second, what we have in the process of Algorithm 2 is a binary tree of size $2P - 3$ each of whose node size is $O(N/P)$. We have obtained an algorithm which has similar effects as m-bridges in $O(N/P + P)$.

The XML representation has such abundance of merits. There is no way to waste these advantages.

B. 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 [31]. The idea proposed is *flattening* of the structures [32]. The importance of this problem domain ignited a lot of researches afterwards, theoretically and in real compilers [33]–[35].

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 list 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 [36], [37]. Chakravarty and Keller extended the structure to involve trees and recursive data structures in general [38], [39]. 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.

C. Parentheses matching

The process of our algorithm development much resembles parentheses matching algorithms, or the All Nearest Smaller Values Problems (ANSVP) which is the generalization of parentheses matching. Their algorithms have been analyzed under CRCW PRAM [7], EREW PRAM [40], hypercube [41] and BSP [8].

The resemblance naturally comes from how tree structures are translated in sequence; the mating pair production in Routine 3 follows the process to find matching of ANSV developed in [7]. We go further to apply evaluation over the data structures. The last paper developed an algorithm which runs $\mathrm{O}(\mathrm{N}/\mathrm{P} + \mathrm{P})$ under distributed memory environments. Our algorithm naturally has complexity comparable to it.

As we have demonstrated the list representations by tree traversals have abundance of advantages. Yet some readers may have concerns on manipulation of our representation—"Isn't this representation vulnerable to insertion or deletion?" Manipulations are easy under linked structures, yet it looks troublesome under the serialized representation.

A solution can be provided by partially reconstructing tree structures. We assume each element in the list has three pointers, namely `prev`, `next`, and `open` if it is close or `close` if open. `prev` and `next` indicates the element before or next to it, and they are immediately obtained from the array indexes. `open` or `close` are to point its corresponding opening or closing element. This pointer information is efficiently obtained using the parentheses matching algorithm.

With the information we can perform every kind of manipulations, like: starting from a white element n, we can locate its next sibling by referring n.`close`.`next`; for inserting a subtree, first create an array to represent the subtree, and modify `next` and `prev` of the elements at the previous and the following position, respectively. When data becomes fragmented, then we reproduce a single array, and apply parallel parentheses matching again after adjusting the data size among processor.

We make a short note that there was an approach under list homomorphism [2]. 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. Our algorithm initially starts from a homomorphic approach, and later loosens the disciplined transactions from the need to create mating pairs.

VIII. Concluding Remarks

This paper presented a new approach for computing trees efficiently in parallel. The essence is to exploit the serialized representation of trees by preorder traversal. We showed that, once tree computations were formatted as tree homomorphism, their parallelized form as Algorithm 1 was obtained at once. This simple idea was not yet free from the factor of tree depth. Later we introduced conditions for parallel tree contractions into tree homomorphism, and developed Algorithm 2 which reduced tree computation into parallel tree contractions of a binary tree. This refined algorithm was free from the depth factor, and its complexity was shown to be $\mathrm{O}(\mathrm{N}/\mathrm{P} + \mathrm{P})$, an almost theoretically optimal result under distributed memory environments. We have implemented a prototype, and the performance it showed was promising.

This research will be improved in the following aspects. Our experiments exhibited good performance toward random inputs. We anticipate the similar analysis by He and Huang [8] will apply to our problems. The flexibility in partitioning our representations was shown to be beneficial under heterogeneous environments with different processor ability. After the mating process, however, data can be reallocated among processors, regardless of their computational power. Theoretical support for these issues are needed.

The framework presented in this paper can work as the kernel of our tree computation libraries. Other tree computations like accumulation using our approach needs consideration.

References

[1] G. L. Miller and J. H. Reif, "Parallel tree contraction and its application," in *Proc. 26th Annual Symposium on Foundations of Computer Science*, Portland, OR, October 1985, pp. 478–489.

[2] M. Cole, "Parallel programming with list homomorphisms." *Parallel Processing Letters*, vol. 5, pp. 191–203, 1995.

[3] S. Gorlatch, "Constructing list homomorphisms," Fakultät für Mathematik und Informatik, Universität Passau, Tech. Rep. MIP-9512, Aug. 1995.

[4] Z. Hu, H. Iwasaki, and M. Takeichi, "Formal derivation of efficient parallel programs by construction of list homomorphisms." *ACM Transactions on Programming Languages and Systems*, vol. 19, no. 3, pp. 444–461, 1997.

[5] E. W. Mayr and R. Werchner, "Optimal tree contraction and term matching on the hypercube and related networks." *Algorithmica*, vol. 18, no. 3, pp. 445–460, 1997.

[6] F. Dehne, A. Ferreira, E. Cáceres, S. Song, and A. Roncato, "Efficient parallel graph algorithms for coarse-grained multicomputers and BSP," *Algorithmica*, vol. 33, no. 2, pp. 183–200, 2002.

[7] O. Berkman, B. Schieber, and U. Vishkin, "Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values," *Journal of Algorithms*, vol. 14, pp. 344–370, 1993.

[8] X. He and C. Huang, "Communication efficient BSP algorithm for all nearest smaller values problem," *Journal of Parallel and Distributed Computing*, vol. 16, pp. 1425–1438, 2001.

[9] J. Hughes, "Why functional programming matters." *Compututer Journal*, vol. 32, no. 2, pp. 98–107, 1989.

[10] R. Bird, *Introduction to Functional Programming using Haskell*. Prentice Hall, Jan. 1998.

[11] L. Valiant, "A bridging model for parallel computation." *Communication of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[12] W. McColl, "Scalable computing." in *Computer Science Today*, ser. Lecture Notes in Computer Science, J. van Leeuwen, Ed. Springer, 1995, vol. 1000, pp. 46–61.

[13] Z. N. Grant-Duff and P. G. Harrison, "Parallelism via homomorphism," *Parallel Processing Letters*, vol. 2, pp. 279–295, 1996.

[14] G. E. Blelloch, "Scans as primitive parallel operations." *IEEE Transactions on Computers*, vol. 38, no. 11, pp. 1526–1538, 1989.

[15] D. B. Skillicorn, "Parallel implementation of tree skeletons." *Journal of Parallel and Distributed Computing*, vol. 39, no. 2, pp. 115–125, 1996.

[16] K. Matsuzaki, Z. Hu, K. Kakehi, and M. Takeichi, "Systematic derivation of tree contraction algorithms," *Parallel Processing Letters*, vol. 15, no. 3, pp. 321–336, 2005, (Original version appeared in *Proc. 4th International Workshop on Constructive Methodology of Parallel Programming*, 2004.).

[17] D. Knuth, *The Art of Computer Programming*, 3rd ed. Addison Wesley, 1997, vol. 1.

[18] W.-N. Chin, A. Takano, and Z. Hu, "Parallelization via context preservation." in *Proc. International Conference on Computer Languages*, 1998, pp. 153–162.

[19] L. Mignet, D. Barbosa, and P. Veltri, "The XML web: a fist study." in *Proceedings of the Twelfth International World Wide Web Conference*. ACM Press, may 2003, pp. 300–510.

[20] E. W. Mayr and R. Werchner, "Optimal routing of parentheses on the hypercube." *Journal of Parallel and Distributed Computing*, vol. 26, no. 2, pp. 181–192, 1995.

[21] SkeTo Project Home Page, `http://www.ipl.t.u-tokyo.ac.jp/sketo/`.

[22] K. Matsuzaki, K. Emoto, H. Iwasaki, and Z. Hu, "A library of constructive skeletons for sequential style of parallel programming (invited paper)," in *Proceedings of the First International Conference on Scalable Information Systems (INFOSCALE 2006)*. IEEE Press, 2006.

[23] M. Reid-Miller, G. L. Miller, and F. Modugno, "List ranking and parallel tree contraction," in *Synthesis of Parallel Algorithms*, J. H. Reif, Ed. Morgan Kaufmann, 1993, ch. 3, pp. 115–194.

[24] A. Gibbons and W. Rytter, "An optimal parallel algorithm for dynamic expression evaluation and its applications," in *Proc. 6th Conference on Foundations of Software Technology and Theoretical Computer Science*, New Delhi, India, 1986, pp. 453–469.

[25] R. Cole and U. Vishkin, "Optimal parallel algorithms for expression tree evaluation and list ranking." in *Proc. 3rd Aegean Workshop on Computing*, ser. Lecture Notes in Computer Science, J. H. Reif, Ed., vol. 319, 1988, pp. 91–100.

[26] K. R. Abrahamson, N. Dadoun, D. G. Kirkpatrick, and T. M. Przytycka, "A simple parallel tree contraction algorithm." *Journal of Algorithms*, vol. 10, no. 2, pp. 287–302, 1989.

[27] R. P. K. Banerjee, V. Goel, and A. Mukherjee, "Efficient parallel evaluation of CSG tree using fixed number of processors," in *Proc. Symposium on Solid Modeling and Applications*, 1993, pp. 137–146.

[28] 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 *Proc. 9th International Conference on High Performance Computing*, ser. Lecture Notes in Computer Science, S. Sahni, V. K. Prasanna, and U. Shukla, Eds., vol. 2552, 2002, pp. 63–78.

[29] J. Sibeyn, "One-by-one cleaning for practical parallel list ranking," *Algorithmica*, vol. 32, no. 3, pp. 345–363, 2002.

[30] K. Matsuzaki, K. Emoto, H. Iwasaki, and Z. Hu, "A library of constructive skeletons for sequential style of parallel programming," submitted to *PPoPP2006*.

[31] G. E. Blelloch, "Programming parallel algorithms." *Communications of the ACM*, vol. 39, no. 3, pp. 85–97, 1996.

[32] G. E. Blelloch and G. Sabot, "Compiling collection-oriented languages onto massively parallel computers." *Journal of Parallel ann Distributed Computing*, vol. 8, no. 2, pp. 119–134, 1990.

[33] D. C. Cann, "Retire fortran? a debate rekindled," *Communication of the ACM*, vol. 35, no. 8, pp. 81–89, 1992.

[34] J. Riely and J. Prins, "Flattening is an improvement," in *Proc. 7th International Symposium on Static Analysis*, ser. Lecture Notes in Computer Science, J. Palsberg, Ed., vol. 1824. Springer, 2000, pp. 360–376.

[35] M. M. T. Chakravarty, G. Keller, R. Lechtchinsky, and W. Pfannenstiel, "Nepal - nested data parallelism in haskell," in *Proc. 7th International Euro-Par Conference*, ser. Lecture Notes in Computer Science, R. Sakellariou, J. Keane, J. R. Gurd, and L. Freeman, Eds., vol. 2150. Springer, 2001, pp. 524–534.

[36] J. Prins and D. W. Palmer, "Transforming high-level data-parallel programs into vector operations." in *Proc. 4th Symposium on Principles & Practice of Parallel Programming*, 1993, pp. 119–128.

[37] D. W. Palmer, J. F. Prins, and S. Westfold, "Work-efficient nested data-parallelism," in *Proc. 5th Symposium on the Frontiers of Massively Parallel Processing*, 1995, pp. 186–193.

[38] G. Keller and M. M. T. Chakravarty, "Flattening trees," in *Proc. 4th International Euro-Par Conference*, ser. Lecture Notes in Computer Science, D. J. Pritchard and J. Reeve, Eds., vol. 1470, 1998, pp. 709–719.

[39] M. M. T. Chakravarty and G. Keller, "More types for nested data parallel programming." in *Proc. 5th International Conference on Functional Programming*, 2000, pp. 94–105.

[40] S. Prasad, S. Das, and C. Chen, "Efficient EREW PRAM algorithms for parentheses-matching," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 9, pp. 995–1008, Sept. 1994.

[41] D. Kravets and C. Plaxton, “All nearest smaller values on the hypercube,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 5, pp. 456–462, Sept. 1996.