Towards Automatic Parallelization of Tree Reductions in Dynamic Programming

Kiminori Matsuzaki kmatsu@ipl.t.u-tokyo.ac.jp
Zhenjiang Hu hu@mist.i.u-tokyo.ac.jp
Masato Takeichi takeichi@mist.i.u-tokyo.ac.jp

Graduate School of Information Science and Technology
University of Tokyo
7-3-1 Hongo, Bunkyo-ku, 113-8656 Tokyo, JAPAN

ABSTRACT

Tree contraction algorithms, whose idea was first proposed by Miller and Reif, are important parallel algorithms to implement efficient parallel programs manipulating trees. Despite their efficiency, the tree contraction algorithms have not been widely used due to the difficulties in deriving the tree contracting operations. In particular, the derivation of the tree contracting operations is much difficult when multiple values are referred and updated in each step of the contractions. Such computations often appear in dynamic programming problems on trees.

In this paper, we propose an algebraic approach to deriving tree contraction programs from recursive tree programs, by focusing on the properties of commutative semirings. We formalize a new condition for implementing tree reductions with the tree contraction algorithms, and give a systematic derivation of the tree contracting operations. Based on it, we implemented a code generator for tree reductions, which has an optimization mechanism that can remove unnecessary computations in the derived parallel programs. As far as we are aware, this is the first step towards an automatic parallelization system for the development of efficient tree programs.

Categories and Subject Descriptors

D.1.3 [**Programming Languages**]: Concurrent Programming—*parallel programming*; D.3.4 [**Programming Languages**]: Processors—*code generation, optimization*

General Terms

Experimentation, Theory

Keywords

Code generator, commutative semiring, dynamic programming, parallel programming, parallel tree contraction, tree

1. INTRODUCTION

In recent years, parallel computing has been getting widely available due to faster and cheaper computers and networks. In scientific parallel computing, regular data structures such as arrays and matrices have been intensively used, and many parallel algorithms and compilation techniques have been developed for these structures. In this paper, we focus on trees, a fundamental data structure widely used in representing general structured data such as XML. As XML is becoming popular, users have their huge data in the form of XML trees, and this situation calls for efficient parallel manipulations of trees.

Developing efficient parallel tree programs is, however, a hard task for programmers because of the ill-balanced structure of trees. In sequential programming, we write programs manipulating trees with recursive functions. For example, we may write the following recursive program for computing the sum of all nodes in a tree.

```
sum_tree(node n) {
  if (n.is_leaf()) {
    return n.v;
  } else {
    return n.v + sum_tree(n.l)
               + sum_tree(n.r);
  }
}
```

In parallel programming, we have to identity computations that can be performed in parallel. As the two recursive calls of `sum_tree` are independent, we may transform this recursive program into a parallel program of the divide-and-conquer style, but the divide-and-conquer program will show poor performance when the input trees are ill-balanced. In addition, we can hardly apply the compilation techniques developed for arrays and matrices, since they work only on loops not on recursive functions.

Fortunately, there is a powerful approach called tree contraction algorithms, first proposed by Miller and Reif [25], which plays an important role in designing efficient parallel programs for trees. The main advantage of the tree contraction algorithms is that the algorithms can always run efficiently in parallel regardless the shape of the input trees. There have been many studies on implementing tree contraction algorithms, on shared memory environments [9, 1, 3], and on distributed memory environments [23, 24, 11].

In spite of a few studies [27, 20] on systematic development (derivation) of parallel tree contraction programs, developing tree contraction programs is still a hard task. To see this, consider a famous dynamic programming problem on trees called "party planning problem" [10].

> The president of a company wants to have a company party. To make the party fun for all attendees, the president does not want both an employee and his or her direct supervisor to attend. The company has a hierarchical structure, that is, the supervisory relations form a tree rooted at the president, and the personnel office has rating each employee with a conviviality rating of a real number. Given the structure of the company and the ratings of employees, the problem is to mark the guests so that the sum of the conviviality ratings of marked guests is its maximum.

To simplify the problem, we assume that the given structure is a binary tree, and compute the maximum sum of the marked guests' ratings. As discussed in [10], we can solve the program by dynamic programming, and write a sequential program in the following recursive way.

```
max_sum(node n) {
  if (n.is_leaf()) {
    return pair(n.v,0);
  } else {
    (l1, l2) = max_sum(n.l);
    (r1, r2) = max_sum(n.r);
    return pair(n.v+l2+r2,
       max(l1,l2) + max\(r1,r2));
  }
}
```

In this program, two values are computed at the same time: the first value (e.g., `l1`) is the maximum sum of marked guests' ratings when the root node of the subtree is marked; the second value (e.g., `l2`) is the maximum sum of marked guests' ratings when the root node is not marked. These simultaneous computations of values make the derivation of tree contraction programs much difficult.

In this paper, we propose an algebraic approach to deriving tree contraction programs from recursive tree programs, by focusing on the properties of commutative semirings. We formalize a condition called *tupled-ring property* to apply the tree contraction approach to the recursive functions with multiple parameters, and show how we can derive tree contraction programs by using this property. We implemented a code generator for translating users' recursive programs into the parallel C++ code based on the derivation algorithm.

The contributions of this paper are summarized as follows.

- We utilized the *properties of algebraic semirings* for deriving parallel programs. So far, associativity and commutativity have been discussed for deriving parallel programs. In this paper, we show distributivity also plays an important role in deriving tree contracting operations.

- We formalize a *new condition* for the dynamic programmings on trees and develop an *algorithm for deriving* the tree contraction programs. Several works gave the conditions for deriving parallel programs, but they did not show how the desired parallel programs can be systematically obtained from the sequential definitions. In contrast, we not only formalize the condition for applying the tree contraction algorithms, but also developed a systematic method for deriving actual tree contraction programs.

- We implemented a *semi-automated code generator* that translates users' annotated recursive (sequential) programs into parallel C++ codes. This system also implements an optimization mechanism that can remove unnecessary computations that may be included when we parallelize the programs.

The paper is organized as follows. In Section 2, we introduce the datatype of binary trees and a general recursive computational pattern called tree reductions, and then review the parallel tree contraction algorithms. In Section 3, we formalize a condition for parallelizing recursive functions based on the properties of commutative semirings, and show the tree contraction programs can be systematically derived from users' recursive programs. In Section 4, we illustrate our code generation system using a non-trivial example of an XPath query. We discuss related work in Section 5, and conclude the paper in Section 6.

2. PRELIMINARIES

2.1 Trees and Tree Reductions

In this paper, we only consider binary trees. A node has two pointer variables, `l` and `r`, which indicate the left and the right child nodes respectively, together with other variables showing information associated to the node (e.g., `v`). To distinguish leaf nodes from internal nodes, we use function `is_leaf()`. In this paper, we borrow a lot of notations of classes in C++ or Java. For instance, we describe the left child of a node `n` as `n.l`.

Generally speaking, algorithms over recursive data structures are often defined recursively along with the definition of the structures. In the case of binary trees, many tree algorithms are specified in the following recursive form, which is called *tree reductions* (or *tree homomorphisms*) [32].

Definition 1. (Tree Reduction) A function `f` is called tree reduction if it is defined with two functions `kl` and `kn` in the following form.

```
f(node n) {
  if (n.is_leaf()) {
    return kl(n.v);
  } else {
    return kn(n.v, f(n.l), f(n.r));
  }
}
```

We denote a tree reduction as $\mathsf{red}(\mathtt{kl}, \mathtt{kn})$. □

The function `sum_tree` in the introduction is a tree reduction $\mathsf{red}(st_l, st_n)$, where the two functions are defined as follows. In the following of the paper, we describes the functions as mathematical equations.

$$\begin{aligned} st_l(v) &= v \\ st_n(v, l, r) &= v + l + r \end{aligned}$$

The function `max_sum` in the introduction is also a tree reduction $\mathsf{red}(ms_l, ms_n)$.

$$\begin{aligned} &ms_l(v) = (v, 0) \\ &ms_n(v, (l_1, l_2), (r_1, r_2)) \\ &\qquad = (v + l_2 + r_2, max(l_1, l_2) + max(r_1, r_2)) \end{aligned}$$

2.2 Tree Contraction Algorithms

The tree contraction algorithms are very important parallel algorithms for efficient tree manipulations. The idea was first introduced by Miller and Reif [25], and later extended with an optimal and practical algorithm on EREW PRAM developed by Abrahamson et al. [1]. Furthermore, implementations on hypercubes and related networks have been developed [23, 24].

The original tree contraction algorithm consists of two primitive operations called *rake* and *compress*. The rake operation merges a leaf with its parent, and the compress operation merges an internal node that has only one child with its child. Several tree contraction algorithms have been developed under the assumption of binary trees. The shunt contraction algorithm developed by Abrahamson et al. [1] uses two symmetric operations instead, namely *contractL* and *contractR*, which are successive calls of the rake operation followed by the compress operation. The contractL operation is applied to a node whose left child is a leaf, and removes two nodes and two edges from the tree as shown in Figure 1. The contractR operation is symmetric to the contractL operation.

Tree contraction algorithms change the order of computations from that of sequential algorithms, and thus some conditions are required to guarantee the correctness of computations. For tree reduction $\mathsf{red}(k_l, k_n)$, a sufficient condition is existence of four auxiliary functions ϕ, ψ_L, ψ_R, and G satisfying the following closure property on G.

$$\begin{aligned} k_n(v, l, r) &= G(\phi(v), l, r) \\ G(v, l, G(v', l', r')) &= G(\psi_L(v, l, v'), l', r') \\ G(v, G(v', l', r'), r) &= G(\psi_R(v, r, v'), l', r') \end{aligned}$$

The first equation says that the function for internal nodes, k_n can be written by an auxiliary function G, which satisfies the following two equations. The latter two equations give the closure property of G with two auxiliary functions ψ_L and ψ_R. If one can develop these four auxiliary functions, then he or she can implement a tree contraction program as shown in Figure 2.

It is worth noting that this condition specifies the equivalent class of acceptable reductions as that of Abrahamson et al. [1], which is given as a closure property on unary functions. We use the condition above in this paper since the condition is more intuitive for the shunt contraction algorithm as the latter two equations exactly correspond to the two contracting operations, namely the contractL and contractR operations.

3. TUPLED-RING PROPERTY

In this section, we give a condition to apply the tree contraction to the reductions that compute multiple values simultaneously. The key idea is to utilize algebraic properties of commutative semirings that the functions have. We also show how we can systematically derive the tree contraction programs from the recursive sequential programs.

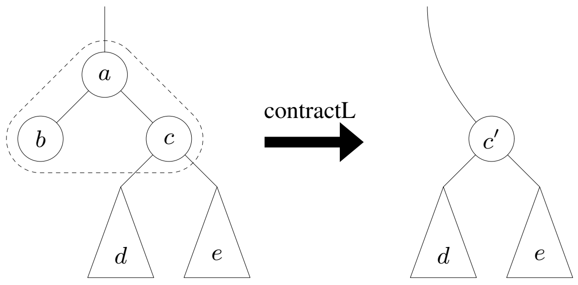

Figure 1: The contractL operation.

1. Number the leaves from left to right starting from 0.
2. Apply ϕ to each internal node and k_l to each leaf.
3. Iterate (a)–(c) until one leaf remains.
   (a) For every internal node whose left child is an even-numbered leaf, apply contractL with function G if the right child is a leaf or with function ψ_L if the right child is an internal node.
   (b) For every internal node that was not involved in the previous step, and whose right child is an even-numbered leaf, apply contractR with function G or ψ_R.
   (c) Renumber leaves by dividing their number by 2.

Figure 2: Shunt contraction algorithm.

First we define commutative semirings.

Definition 2. An algebra $\mathcal{A} = \{\mathcal{D}, \oplus, \otimes\}$ is said to be a commutative semiring, if $\mathcal{D}$ is the carrier, $\oplus$ is an associative and commutative operator with unit $\iota_\oplus$, and $\otimes$ is an associative and commutative operator with unit $\iota_\otimes$ and distributes over $\oplus$. $\square$

Three examples of commutative semirings are

$$\{\mathbf{Num}, +, \times\}, \\ \{\mathbf{Num}, \uparrow, +\}, \text{and} \\ \{\mathbf{Bool}, \vee, \wedge\}$$

where the operator $\uparrow$ denotes the *max* function that returns the bigger of the two inputs. The latter two commutative semirings are often seen in dynamic programmings.

Next, we define a class of functions on these commutative semirings. For some finite k, $\mathcal{D}^k$ denotes a set of finitely tupled values $(v_1, v_2, \ldots, v_k)$ where $v_i \in \mathcal{D}$.

Definition 3. Let $\{\mathcal{D}, \oplus, \otimes\}$ be a commutative semiring. Function $g :: \mathcal{D}^k \to \mathcal{D}$ is said to be a *linear polynomial function*, if it can be defined in the following form:

$$\begin{aligned} &g\ (x_1, x_2, \ldots, x_k) \\ &\quad = (a_1 \otimes x_1) \oplus (a_2 \otimes x_2) \oplus \cdots \oplus (a_k \otimes x_k) \oplus a_{k+1} \end{aligned}$$

where $a_1, a_2, \ldots, a_k$ and a_{k+1} are constants. $\square$

The function for internal nodes in a tree reduction takes two sets of recursive results from the left child and the right child. For example, the function ms_n in Section 2.1 takes (l_1, l_2) and (r_1, r_2). Therefore, we specify a class of functions that take two sets of values on an analogy of linear polynomial functions.

Definition 4. (Bi-linear Polynomial Function) Let $\mathcal{A}$ be a domain and $\{\mathcal{D}, \oplus, \otimes\}$ be a commutative semiring. Function $g :: (\mathcal{A}, \mathcal{D}^k, \mathcal{D}^k) \to \mathcal{D}$ is said to be a *bi-linear polynomial function*, if it can be defined in the following two forms:

$$\begin{aligned}&g\ (v, (l_1, l_2, \ldots, l_k), (r_1, r_2, \ldots, r_k))\\&\quad = (a_1 \otimes l_1) \oplus (a_2 \otimes l_2) \oplus \cdots \oplus (a_k \otimes l_k) \oplus a_{k+1}\\&\quad = (b_1 \otimes r_1) \oplus (b_2 \otimes r_2) \oplus \cdots \oplus (b_k \otimes r_k) \oplus b_{k+1}\end{aligned}$$

where $a_1, a_2, \ldots, a_k$ and a_{k+1} are values computed only from $r_1, r_2, \ldots, r_k$ and v; $b_1, b_2, \ldots, b_k$ and b_{k+1} are values computed only from $l_1, l_2, \ldots, l_k$ and v. $\square$

Note that the class of the bi-linear polynomial functions is broader than that of the linear functions with respect to all the arguments. For example, the following function g

$$g(v, (l_1), (r_1)) = l_1 \otimes r_1$$

is a bi-linear polynomial function but not a linear polynomial function with respect to l_1 and r_1.

In this paper, we deal with the reductions whose function for internal nodes is defined with k bi-linear polynomial functions. In the following of this section, we use the function ms_n for the party planning problem as our running example.

$$\begin{aligned}&ms_n(v, (l_1, l_2), (r_1, r_2))\\&\quad = \begin{pmatrix} v + l_2 + r_2 \\ (l_1 \uparrow l_2) + (r_1 \uparrow r_2) \end{pmatrix}\end{aligned}$$

Let ms_{n1} and ms_{n2} be functions that computes the first and second results of the function ms_n. These two functions are indeed bi-linear polynomial functions as seen in the following transformations. Note that $-\infty$ is the unit of operator $\uparrow$, and is the zero-element on $\{\mathbf{Num}, \uparrow, +\}$.

$$\begin{aligned}&ms_{n1}(v, (l_1, l_2), (r_1, r_2))\\&\quad = (-\infty + l_1) \uparrow ((v + r_2) + l_2) \uparrow -\infty\\&\quad = (-\infty + r_1) \uparrow ((v + l_2) + r_2) \uparrow -\infty\end{aligned}$$

$$\begin{aligned}&ms_{n2}(v, (l_1, l_2), (r_1, r_2))\\&\quad = ((r_1 \uparrow r_2) + l_1) \uparrow ((r_1 \uparrow r_2) + l_2) \uparrow -\infty\\&\quad = ((l_1 \uparrow l_2) + r_1) \uparrow ((l_1 \uparrow l_2) + r_2) \uparrow -\infty\end{aligned}$$

We have observed several examples that can be parallelized by the tree contraction algorithms (some examples are in [21]), and would like to conjecture that *a reduction algorithm* $\mathsf{red}(k_l, k_n)$ *could be parallelized by the tree contraction algorithms, if the function* k_n *is defined with a set of bi-linear polynomial functions.* In the rest of this section, we will show that this conjecture is true by deriving all four auxiliary functions from the definition of the function k_n.

The key idea is that the set of linear polynomial functions can be represented with matrices and matrix multiplications. The set of k bi-linear polynomial functions are formalized as the following (k+1)-dimensional matrix multiplications on the semiring $\{\mathcal{D}, \oplus, \otimes\}$. Let $x_1, x_2, \ldots, x_k$ be results of function k_n, that is,

$$(x_1, x_2, \ldots, x_k) = k_n(v, (l_1, l_2, \ldots, l_k), (r_1, r_2, \ldots, r_k))$$

then we can formalize the computation of k_n as the following (k+1)-dimensional matrix multiplications on the commutative semiring $\{\mathcal{D}, \oplus, \otimes\}$. For readability, we may denote a tuple as a column vector.

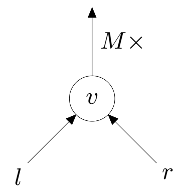

Figure 3: Intuitive meaning of G and the assigned values v and M. At the internal node we compute $k_n(v, \boldsymbol{l}, \boldsymbol{r})$ followed by the multiplication of M.

$$\begin{aligned}&(x_1, \ldots, x_k, \iota_\otimes)^T\\&= \begin{pmatrix} a_{11} & \cdots & a_{1k} & a_{1(k+1)} \\ \vdots & \ddots & \vdots & \vdots \\ a_{k1} & \cdots & a_{kk} & a_{k(k+1)} \\ \iota_\oplus & \cdots & \iota_\oplus & \iota_\otimes \end{pmatrix} \times_{\otimes,\oplus} \begin{pmatrix} l_1 \\ \vdots \\ l_k \\ \iota_\otimes \end{pmatrix}\\&= \begin{pmatrix} b_{11} & \cdots & b_{1k} & b_{1(k+1)} \\ \vdots & \ddots & \vdots & \vdots \\ b_{k1} & \cdots & b_{kk} & b_{k(k+1)} \\ \iota_\oplus & \cdots & \iota_\oplus & \iota_\otimes \end{pmatrix} \times_{\otimes,\oplus} \begin{pmatrix} r_1 \\ \vdots \\ r_k \\ \iota_\otimes \end{pmatrix}\end{aligned}$$

In the following, we denote the matrices and tupled values in the bold font such as $\boldsymbol{M}$ or $\boldsymbol{l}$, and $\times$ for the matrix multiplication. By using this matrix notation, we can rephrase the condition for the tree contraction algorithms. Let $\boldsymbol{l} = (l_1, l_2, \ldots, l_k)^T$, $\boldsymbol{r} = (r_1, r_2, \ldots, r_k)^T$, $g_r(v, \boldsymbol{r})$ be the matrix of $\{a_{ij}\}$ above, and $g_l(v, \boldsymbol{l})$ be the matrix of $\{b_{ij}\}$. With the two functions g_l and g_r, we can specify the condition simply as follows.

$$\begin{pmatrix} k_n(v, \boldsymbol{l}, \boldsymbol{r}) \\ \iota_\otimes \end{pmatrix} = g_r(v, \boldsymbol{r}) \times \begin{pmatrix} \boldsymbol{l} \\ \iota_\otimes \end{pmatrix}$$

$$\begin{pmatrix} k_n(v, \boldsymbol{l}, \boldsymbol{r}) \\ \iota_\otimes \end{pmatrix} = g_l(v, \boldsymbol{l}) \times \begin{pmatrix} \boldsymbol{r} \\ \iota_\otimes \end{pmatrix}$$

For our running example, ms_n, we can derive the functions g_l and g_r as follows straightforwardly from the definition of bi-linear polynomial functions.

$$g_l\left(v, \begin{pmatrix} l_1 \\ l_2 \end{pmatrix}\right) = \begin{pmatrix} -\infty & v + l_2 & -\infty \\ l_1 \uparrow l_2 & l_1 \uparrow l_2 & -\infty \\ -\infty & -\infty & 0 \end{pmatrix}$$

$$g_r\left(v, \begin{pmatrix} r_1 \\ r_2 \end{pmatrix}\right) = \begin{pmatrix} -\infty & v + r_2 & -\infty \\ r_1 \uparrow r_2 & r_1 \uparrow r_2 & -\infty \\ -\infty & -\infty & 0 \end{pmatrix}$$

We now develop the four auxiliary functions from the functions k_n, g_r, and g_l given in the matrix representation. We assign for each node the original value of the node v and a (k+1)-dimension matrix $\boldsymbol{M}$, and define the auxiliary function G as the following equation.

$$\begin{pmatrix} G((v, \boldsymbol{M}), \boldsymbol{l}, \boldsymbol{r}) \\ \iota_\otimes \end{pmatrix} = \boldsymbol{M} \times \begin{pmatrix} k_n(v, \boldsymbol{l}, \boldsymbol{r}) \\ \iota_\otimes \end{pmatrix}$$

An intuitive meaning of G is illustrated in Figure 3.

As discussed in Section 2, a condition for the tree contraction algorithms is given by three equations. After the substitution of $(v, \boldsymbol{M})$ to v, the first equation is given as

$$k_n(v, \boldsymbol{l}, \boldsymbol{r}) = G(\phi(v), \boldsymbol{l}, \boldsymbol{r})$$

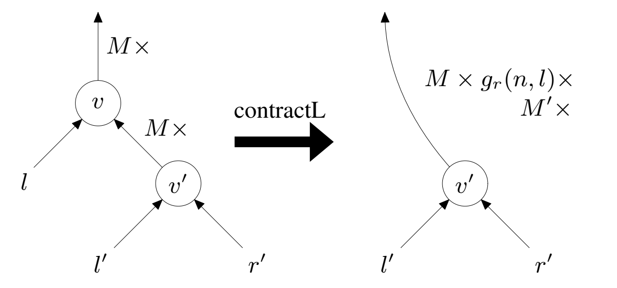

Figure 4: Contracting operation on matrix notation.

and by simple calculations with the definition of G, we obtain the definition of ϕ:

$$\phi(v) = (v, \boldsymbol{I})$$

where $\boldsymbol{I}$ is the identity matrix of $(k+1)$-dimension. The identity matrix is a matrix whose diagonal elements have value $\iota_{\otimes}$ and the other elements have value $\iota_{\oplus}$.

Next, we derive the auxiliary functions for the contracting operation, ψ_L and ψ_R, from the latter two equations in the conditions of the tree contraction. When the left child is a leaf we apply the contractL operation, for which we require the following equation on auxiliary functions.

$$\begin{aligned} &G((v,\boldsymbol{M}), \boldsymbol{l}, G((v',\boldsymbol{M'}), \boldsymbol{l'}, \boldsymbol{r'})) \\ &\quad = G(\psi_L((v,\boldsymbol{M}), \boldsymbol{l}, (v',\boldsymbol{M'})), \boldsymbol{l'}, \boldsymbol{r'}) \end{aligned}$$

We calculate the left-hand side as follows.

$$\begin{aligned} &\begin{pmatrix} G((v,\boldsymbol{M}), \boldsymbol{l}, G((v',\boldsymbol{M'}), \boldsymbol{l'}, \boldsymbol{r'})) \\ \iota_{\otimes} \end{pmatrix} \\ &\quad = \{\text{Definition of } G\} \\ &\quad\quad \textbf{let} \begin{pmatrix} \boldsymbol{r} \\ \iota_{\otimes} \end{pmatrix} = \boldsymbol{M'} \times \begin{pmatrix} k_n(v', \boldsymbol{l'}, \boldsymbol{r'}) \\ \iota_{\otimes} \end{pmatrix} \\ &\quad\quad \textbf{in } \boldsymbol{M} \times \begin{pmatrix} k_n(v, \boldsymbol{l}, \boldsymbol{r}) \\ \iota_{\otimes} \end{pmatrix} \\ &\quad = \{\text{Application of } g_l \text{ to the parent}\} \\ &\quad\quad \boldsymbol{M} \times g_l(v, \boldsymbol{l}) \times \boldsymbol{M'} \times \begin{pmatrix} k_n(v', \boldsymbol{l'}, \boldsymbol{r'}) \\ \iota_{\otimes} \end{pmatrix} \\ &\quad = \{\text{Associativity of } \times\text{, and definition of } G\} \\ &\quad\quad \begin{pmatrix} G((v', \boldsymbol{M} \times g_l(v,\boldsymbol{l}) \times \boldsymbol{M'}), \boldsymbol{l'}, \boldsymbol{r'}) \\ \iota_{\otimes} \end{pmatrix} \end{aligned}$$

From the calculations above, we obtain the definition of the auxiliary function ψ_L as follows, whose intuitive meaning is shown in Figure 4.

$$\psi_L((v,\boldsymbol{M}), \boldsymbol{l}, (v',\boldsymbol{M'})) = (v', \boldsymbol{M} \times g_l(v,\boldsymbol{l}) \times \boldsymbol{M'})$$

Symmetric to ψ_L, we can derive the auxiliary function ψ_R as follows.

$$\psi_R((v,\boldsymbol{M}), \boldsymbol{r}, (v',\boldsymbol{M'})) = (v', \boldsymbol{M} \times g_r(v,\boldsymbol{r}) \times \boldsymbol{M'})$$

We have successfully derived all the auxiliary functions for the tree contraction algorithms, ϕ, ψ_L, ψ_R, and G, and now summarize the discussion as the following theorem.

THEOREM 1. *Let k_l be a function and k_n be a function defined with a set of bi-linear functions. Reduction algorithm* $\mathsf{red}(k_l, k_n)$ *can be parallelized by using the tree contraction algorithms.*

PROOF. We can derive the auxiliary functions from the function k_n as discussed so far, and these auxiliary functions guarantee the parallel computation based on the tree contraction algorithms. $\square$

For our running example, we can derive the four auxiliary functions for the tree contraction algorithms as shown in Figure 5, by simply substituting the definitions of I, g_l, etc. to those in the results of the derivation so far. We omitted the elements on the third column and the elements on the third row, since they do not change their values throughout the tree contracting computation. We discuss how we can automatically remove these values in the following section.

4. CODE GENERATOR

The tupled-ring property in the previous section gives a clear condition for parallelizing reductions with multiple parameters, but developing suitable functions is somehow tedious due to the large number of parameters introduced in the matrices.

To encourage programmers to develop parallel programs based on the tupled-ring property, we have developed a prototype system that automatically translates users' recursive specifications into parallel C++ codes. In this section, we describe the outline of our code generator, and then demonstrate the parallelization steps with a non-trivial example.

4.1 Outline of Code Generator

Figure 6 depicts the outline of our code generator. It takes recursive functions written in C++ like notation with some annotations for its input (Figure 10). This notation includes not only operations and functions but also `if`-statements. We introduced a notation for tuples to enable users to write concisely tree algorithms computing multiple values. We ask users to specify the properties among operators (i.e., commutative semirings) as annotations. For example, users should specify the operators and the units that construct a commutative semiring as shown in the first line in Figure 10.

Our code generator first splits the specification into two parts corresponding to two cases for leaves and for internal nodes by finding `if`-statement with the predicate of `is_leaf()`. The system then transforms the definition for internal nodes into the canonical forms to generate the matrices for functions g_l and g_r. This canonicalization is performed in the following three steps.

1. We expand the expressions by using the distributive law $x \otimes (b \oplus c) = (x \otimes b) \oplus (x \otimes c)$. Note that the `if`-statement distributes over any operations.

2. We flatten the expression with the associative law, and sort the arguments with the commutative law.

3. We put sub-expressions together for each argument by using the distributive law in the reversed direction. Here if there is no occurrence of an argument x_i, then we insert $(\iota_{\oplus} \otimes x_i)$ that is equal to $\iota_{\oplus}$.

After canonicalization, the system checks whether each expression is a linear function, and generates the matrices for two functions g_l and g_r.

After deriving the matrices, the system proceeds into the optimization phase. In the optimization phase, the system abstracts the values to four values `Z`, `I`, `C` and `V`:

$$
\begin{aligned}
\phi(v) &= \left(v, \begin{pmatrix} 0 & -\infty \\ -\infty & 0 \end{pmatrix}\right) \\
\psi_L\left(\left(v, \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}\right), \begin{pmatrix} l_1 \\ l_2 \end{pmatrix}, \left(v', \begin{pmatrix} a'_{11} & a'_{12} \\ a'_{21} & a'_{22} \end{pmatrix}\right)\right) &= \left(v', \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \times \begin{pmatrix} -\infty & v + l_2 \\ l_1 \uparrow l_2 & l_1 \uparrow l_2 \end{pmatrix} \times \begin{pmatrix} a'_{11} & a'_{12} \\ a'_{21} & a'_{22} \end{pmatrix}\right) \\
\psi_R\left(\left(v, \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}\right), \begin{pmatrix} r_1 \\ r_2 \end{pmatrix}, \left(v', \begin{pmatrix} a'_{11} & a'_{12} \\ a'_{21} & a'_{22} \end{pmatrix}\right)\right) &= \left(v', \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \times \begin{pmatrix} -\infty & v + r_2 \\ r_1 \uparrow r_2 & r_1 \uparrow r_2 \end{pmatrix} \times \begin{pmatrix} a'_{11} & a'_{12} \\ a'_{21} & a'_{22} \end{pmatrix}\right) \\
G\left(\left(v, \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}\right), \begin{pmatrix} l_1 \\ l_2 \end{pmatrix}, \begin{pmatrix} r_1 \\ r_2 \end{pmatrix}\right) &= \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \times \begin{pmatrix} v + l_2 + r_2 \\ (l_1 \uparrow l_2) + (r_1 \uparrow r_2) \end{pmatrix}
\end{aligned}
$$

Figure 5: The definition of contracting operations for the party planning problem. The operator $\times$ is the matrix multiplication on the commutative semiring $\{\mathbf{Num}, \uparrow, +\}$.

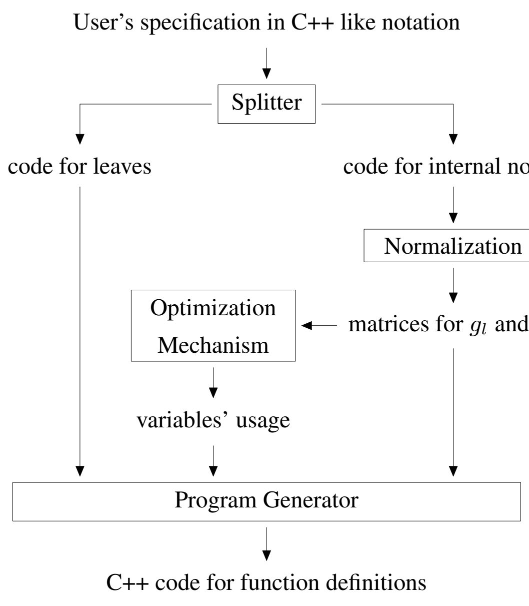

Figure 6: Outline of our code generator.

- a `Z` element denotes the zero-element of the commutative semiring ($= \iota_\oplus$);
- an `I` element denotes the identity-unit of the commutative semiring ($= \iota_\otimes$);
- a `C` element denotes a constant value;
- a `V` element denotes a non-constant value.

First, the system compares the corresponding values in the matrices for g_l, g_r and the identity matrix, and generates an initial matrix for the analysis. In this initial matrix, the `V` elements denote that the values on the positions are required to the tree contraction algorithms. The system then simulates the computations of the tree contracting operations, by squaring the matrix using the operators given in Figure 7 until the matrix does not change. Computations in squaring have different semantics from original $\oplus$ and $\otimes$: for example, on the original algebra $\mathtt{Z} \otimes \mathtt{0} = \mathtt{Z}$ holds, but in the optimization phase we consider $\mathtt{Z} \otimes' \mathtt{0} = \mathtt{V}$ since the inputs and the output differ. Note that the iteration terminates, since during the squaring the matrix the value may change only to `V`, and once an element has the value `V` then the value never changes any more. In the result matrix,

$\oplus'$	Z	I	C	V
Z	Z	I	C	V
I	I	V	V	V
C	C	V	V	V
V	V	V	V	V

$\otimes'$	Z	I	C	V
Z	Z	V	V	V
I	V	I	C	V
C	V	C	V	V
V	V	V	V	V

Figure 7: Semantics of two operations on four values.

the value `V` indicates that the element should be computed through the tree contraction because the value may change; and the other values denote that the elements do not change through the tree contractions and we can remove them by substituting the values to the variables. If the value is `Z` or `I` then we can furthermore remove the computations as well. Thus, this optimization can reduce the computation time as well as the memory space during the tree contractions.

The system finally generates the code of parallel programs. Since tree contraction algorithms have quite different implementations on various parallel environments, it is unrealistic to generate the program code specific to each lower-level architecture. The system, therefore, generates the code for the parallel tree skeletons in the SkeTo library [19]. The parallel tree skeletons abstract the lower-level tree contraction algorithms and we can obtain a reasonably fast parallel code by supplying the definition of parameter functions of the skeletons. The system generates C++ code for the definition of tuples, the function objects for k_l, k_n, ϕ, ψ_L, ψ_R and G.

4.2 Example: Parallelizing XPath Queries

In this section, we demonstrate our code generator with a nontrivial application, namely parallelization of XPath queries. XPath query [5] is one of the core processings in the XSLT [17] and XQuery [6], which is widely used in processing XML trees. As our running example, we will generate a parallel code for the following XPath query.

```
//a[./b/following-sibling::c]
```

This XPath query searches a node labeled `a` that has children labeled `b` and `c` in this order from left (Note that other children may appear).

Since XML tree is a general tree whose internal nodes may have an arbitrary number of children, we need to represent a XML tree as a binary tree. Here, we use a binary-tree representation illustrated in Figure 8 [10]. In this binary-tree representation, the left child and the right child of a node denote the left-most child and the right sibling in the original XML tree, respectively.

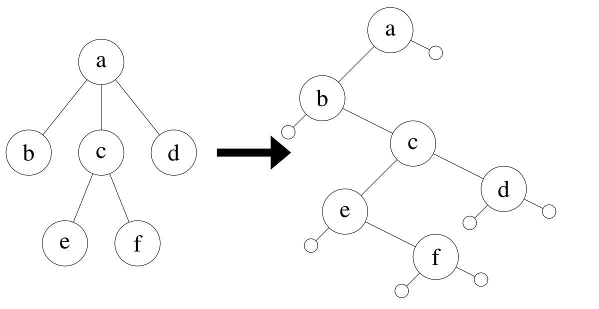

Figure 8: Binary-tree representation of XML tree.

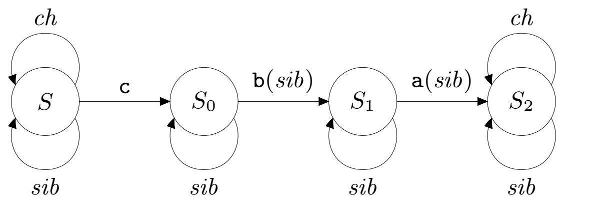

Figure 9: A nondeterministic automaton for the sample XPath query. Transition labeled $x(ch)$ occurs if the child has the original state and the node has value x. Transition labeled $x(sib)$ shows the case of sibling.

It is known that an XPath query can be translated into an automaton [29]. In the case of our running example, the XPath query is performed by matching the nondeterministic finite automaton shown in Figure 9 to the path from each node to the root. By using dynamic programming technique, we can write a recursive program that performs the XPath query by computing three variables `v0`, `v1` and `v2` (Figure 10), which respectively correspond to the three states S_0, S_1 and S_2 in Figure 9.

Now we demonstrate how the system generates a parallel program from the input sequential program based on the tupled-ring property.

The system first splits the definitions for leaves and internal nodes, and parses the definition for internal nodes. In this analysis, the system generates an abstract syntax tree corresponding to the following segment of program.

```
//semiring (bool, ||, &&, false, true)
//recursion (l0, l1, l2) (r0, r1, r2)
//node      v
//results   (v0, v1, v2)
v0 = r0 || (v == 'c')
v1 = ((v == 'b') && r0) || r1
v2 = ((v == 'a') && l1) || l2 || r2
```

The system then normalizes the abstract syntax tree for each equation into the canonical forms. Here, let us consider normalizing the equation of `v2` for the parameters `r0`, `r1` and `r2`. At the first phase, the system applies the distributive law to expand all the parts related to the parameters. For the equation of `v2` the system does nothing, since the operator `&&` is inside of `||`. The system then sorts the subexpressions with respect to the arguments `r0`, `r1` and `r2`.

```
v2 = ((v == 'a') && l1) || l2 || r2
   = r2 || ((v == 'a') && l1) || l2
```

Finally, the system applies the distributive law in the reversed direction. For the arguments `r0` and `r1`, there are no occurrences of the arguments and thus the system inserts a special value `ZERO` representing $\iota_\oplus$ with the arguments. For the argument `r2`, there is no coefficient and thus the system inserts a special value `ONE` representing $\iota_\otimes$. Therefore, the system transforms the equation as follows.

```
v2 = r2 || ((v == 'a') && l1) || l2
   = (ZERO && r0)
     (ZERO && r1) ||
     (ONE  && r2) ||
     (((v == 'a') && l1) || l2)
```

After normalizing all the expressions with respect to the parameters `r0`, `r1` and `r2`, we obtain the following 4×4 matrix for the function g_l.

```
ONE       ZERO  ZERO  (v=='c')
(v=='b')  ONE   ZERO  ZERO
ZERO      ZERO  ONE   ((v=='a')&&l1)||l2
ZERO      ZERO  ZERO  ONE
```

In the same way, we obtain the following matrix for the function g_r.

```
ZERO  ZERO      ZERO  r0||(v=='c')
ZERO  ZERO      ZERO  ((v='b')&&r0)||r1
ZERO  (v=='a')  ONE   r2
ZERO  ZERO      ZERO  ONE
```

Of course, the identity matrix is given as follows.

```
ONE   ZERO  ZERO  ZERO
ZERO  ONE   ZERO  ZERO
ZERO  ZERO  ONE   ZERO
ZERO  ZERO  ZERO  ONE
```

By comparing these three matrices, the system generates a matrix for the optimization phase. In this comparison, if all the corresponding values are `ZERO`s the value in the generated matrix becomes `Z`, and if the corresponding values are different the value in the generated matrix becomes `V` even if values are `Z` or `I`. Of course, if an element has variables such as `v`, `l0` and `r0`, then the corresponding element necessarily becomes `V`. For our example, we obtain the following matrix as the initial matrix of the optimization phase.

```
V Z Z V
V V Z V
Z V I V
Z Z Z I
```

In the optimization phase, the system iterates squaring the matrix until the same matrix appears, and the calculation yields the following results.

```
V Z Z V       V Z Z V       V Z Z V
V V Z V  -->  V V Z V  -->  V V Z V
Z V I V       V V I V       V V I V
Z Z Z I       Z Z Z I       Z Z Z I
```

The last matrix above has eight `V` elements, and indicates that we need those eight elements in the computations of tree contracting operations. The other values can be removed from the generated parallel code, and thus we can reduce the number of variables to a half by this optimization.

```
// semiring(bool, ||, &&, false, true);
tuple<bool> xpquery(node< char > n) {
  if (n.is_leaf()) {
    return (false, false, false);
  } else {
    tuple<bool> (l0, l1, l2) = xpquery(n.l);
    tuple<bool> (r0, r1, r2) = xpquery(n.r);
    bool v0 = r0 || (n.v == 'c');
    bool v1 = ((n.v == 'b') && r0) || r1;
    bool v2 = ((n.v == 'a') && l1) || l2 || r2;
    return (v0, v1, v2);
  }
}
```

Figure 10: A sample code of input specification.

```
struct xpquery_inter_val
{
  char v; bool a_0_0, a_0_3, a_1_0, a_1_1, a_1_3, a_2_0, a_2_1, a_2_3;
  xpquery_inter_val() {
    a_0_0 = true;  a_0_3 = false; a_1_0 = false; a_1_1 = true;
    a_1_3 = false; a_2_1 = false; a_2_0 = false; a_2_3 = false;
  }
};
  ...
struct xpquery_psi_L {
  xpquery_inter_val operator()( xpquery_inter_val n, xpquery_ret_val l, xpquery_inter_val r ) {
    xpquery_inter_val res;
    char v = n.v;
    res.a_0_0 = n.a_0_0 && r.a_0_0;
    res.a_0_3 = (n.a_0_0 && (r.a_0_3 || (v == 'c'))) || n.a_0_3;
    res.a_1_0 = (n.a_1_1 && r.a_1_0) || (((n.a_1_1 && (v == 'b')) || n.a_1_0) && r.a_0_0);
       ....
    res.v = r.v;
    return res;
  }
};
  ...
/* in user's code */
    ret_val n = tree_skeletons::reduce(
                    xpquery_leaf(), xpquery_node(), xpquery_phi(),
                    xpquery_psi_L(), xpquery_psi_R(), xpquery_G(), tree);
```

Figure 11: Segments of generated code for the sample XPath query.

The system finally generates the parallel program in C++ code. The system first generates the definition of structures for result tuples and for internal contraction steps based on the matrix obtained so far. In our running example, the structure for result tuples `xpquery_ret_val` has three variables `v0`, `v1` and `v2`, and the structure for the contraction `xpquery_inter_val` has eight values corresponding to the `V` elements. The system then generates the definition of function objects for functions k_l and k_n and auxiliary functions ϕ, ψ_L, ψ_R, and G, from the definition of leaf's case and the two matrices of internal node's case. Segments of the generated code are shown in Figure 11.

5. RELATED WORK

Tree Contraction Algorithms

Tree contraction algorithms, whose idea was first proposed by Miller and Reif [25], are very important parallel algorithms for trees. Many researchers have devoted themselves to developing efficient implementations on various parallel models [1, 3, 4, 9, 13, 23, 24]. Among them, Gibbons and Rytter developed an cost-optimal algorithm on CREW PRAM [13]; Abrahamson et al. developed an cost-optimal and practical algorithm on EREW PRAM [1]; Miller and Reif showed implementations on hypercubes or related networks [23, 24]. The parallel programs derived based on the tupled-ring property can be implemented with all of them.

A lot of tree programs have been described by the tree contraction algorithms [4, 9, 13, 15, 20, 26, 27, 28]. Many of these programs, however, compute a single value instead of tupled values at each contraction step. For example, Cole and Vishkin [9] and He [15] developed parallel algorithms based on the finiteness of the domain, e.g. for the minimum covering-set problem and the maximum independent-set problem. Though the maximum independent-set problem is a simpler version of the party planning problem, their approaches are not applicable to development of parallel programs of the party planning problem. Miller and Teng [27] proposed a method for developing parallel programs on computational trees with *min* and *max* functions by focusing on the algebraic properties. They also extended their idea to the evaluation of computational circuits (trees whose each leaf has a value and each internal node has an operator) with finite-sized matrices [26, 28]. Though their approaches are interesting in theory, they impose much restriction on the operators associated to each internal node. We give a concise and practical condition for tree contraction where we can define any computations on the internal nodes under the flexible condition.

Automatic and Systematic Parallelization

Automatic parallelization of programs is a quite big challenge, and there have been several studies on automatic parallelization of loops over arrays. Fisher and Ghuloum [12] have developed a parallelization system for loops on arrays based on the isomorphism on the shape of program code. Lu and Mellor-Crummey [18] have developed more powerful pattern matching and code generation mechanisms on distributed memory environments. Xu et al. [33] have focused on not only associativity but also distributivity to derive parallel programs from users' programs on lists or arrays, and developed an automatic parallelization system. These studies succeeded in generating automatically the efficient parallel code from users' sequential code. Though there are several studies on parallelizing loops, as far as we are aware there is no (semi-)automatic parallelization system that can be applied to tree structures.

Our work is also related to systematic derivation of parallel programs. Systematic parallelization has been actively studied in the framework of *skeletal parallel programming* [8], and many studies have been done [7, 14, 16, 22, 30] on lists or arrays. For trees, several researchers have studied the systematic parallelization. Skillicorn formalized five primitive computational patterns [31], and the tree reduction is one of them. Ahn and Han [2] and we [21] have developed systematic methods for decomposing complex recursive programs into the combinations of the primitive patterns. Our work is addressed to the generation of efficient parallel programs from the primitives derived so far.

6. CONCLUSION

In this paper, we have proposed a concise condition named tupled-ring property for developing tree contraction programs for complex reductions. The key idea is to focus on the algebraic properties of commutative semirings. We have demonstrated application of the tupled-ring property to several dynamic programming problems on trees in the paper. We have developed a code generation system based on this tupled-ring property, with an optimization mechanism that removes unnecessary computations inserted in applying the tupled-ring property. The system can automatically transform users' recursive reduction programs with some annotations into parallel programs. With the tupled-ring property and the code generation system, we can develop parallel programs manipulating trees more easily from the familiar sequential specifications.

One of our future work are to extend the pattern matching routine for other tree manipulating algorithms such as accumulations or scans to support the five primitives of parallel tree skeletons.

Acknowledgments

We would like to thank Noriyuki Ohkawa for his help in implementing the code generation system.

This work was supported by the Grant-in-Aid for Scientific Research (B) No. 17300005 from Japan Society for the Promotion of Science, and the Grant-in-Aid for Young Scientists (B) No. 18700021 from the Ministry of Education, Culture, Sports, Science and Technology.

7. REFERENCES

[1] K. Abrahamson, N. Dadoun, D. G. Kirkpatrik, and T. Przytycka. A simple parallel tree contraction algorithm. *Journal of Algorithms*, 10(2):287–302, June 1989.

[2] J. Ahn and T. Han. An analytical method for parallelization of recursive functions. *Parallel Processing Letters*, 10(1):87–98, 2000.

[3] 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). In *9th International Conference on High Performance Computing (HiPC 2002)*, volume 2552 of *Lecture Notes in Computer Science*, pages 63–75, Bangalore, India, December 2002.

[4] R. P. K. Banerjee, V. Goel, and A. Mukherjee. Efficient parallel evaluation of CSG tree using fixed number of processors. In *ACM Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 137–146, Montreal, Canada, May 1993.

[5] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Simeon, editors. *XML Path Language (XPath) 2.0*. W3C Working Draft 29, 2004. Available from `http://www.w3.org/TR/xpath20/`.

[6] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon, editors. *XQuery 1.0: An XML Query Language*. W3C Working Draft 29, 2004. Available from `http://www.w3.org/TR/xquery/`.

[7] W. N. Chin, A. Takano, and Z. Hu. Parallelization via context preservation. *IEEE Computer Society International Conference on Computer Languages (ICCL'98)*, pages 153–162, May 1998.

[8] M. Cole. *Algorithmic skeletons : A structured approach to the management of parallel computation.* Research Monographs in Parallel and Distributed Computing, Pitman, London, 1989.

[9] R. Cole and U. Vishkin. The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time. *Algorithmica*, 3:329–346, 1988.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[11] F. Dehne, A. Ferreira, E. Caceres, S. W. Song, and A. Roncato. Efficient parallel graph algorithms for coarse grained multicomputers and BSP. *Algorithmica*, 33(2):183–200, 2002.

[12] A. L. Fisher and A. M. Ghuloum. Parallelizing complex scans and reductions. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI '94)*, pages 135–146, Orlando, June 1994.

[13] A. Gibbons and W. Rytter. An optimal parallel algorithm for dynamic expression evaluation and its applications. In *Proceedings of the sixth conference on Foundations of software technology and theoretical computer science*, pages 453–469, New Delhi, India, 1986. Springer-Verlag New York, Inc.

[14] S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In *Annual European Conference on Parallel Processing, LNCS 1124*, pages 401–408, LIP, ENS Lyon, France, August 1996. Springer-Verlag.

[15] X. He. Efficient parallel algorithms for solving some tree problems. In *24th Allerton Conference on Communication, Control and Computing*, pages 777–786, 1986.

[16] Z. Hu, M. Takeichi, and H. Iwasaki. Diffusion: Calculating efficient parallel programs. In *1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '99)*, pages 85–94, San Antonio, Texas, January 1999. BRICS Notes Series NS-99-1.

[17] M. Kay, editor. *XSL Transformations (XSLT) Version 2.0.* W3C Working Draft 5, 2004. Available from `http://www.w3.org/TR/xslt20/`.

[18] B. Lu and J. Mellor-Crummey. Compiler optimization of implicit reductions for distributed memory multiprocessors. In *Proceedings of the International Parallel Processing Symposium*, 1998.

[19] K. Matsuzaki, K. Emoto, H. Iwasaki, and Z. Hu. A library of constructive skeletons for sequential style of parallel programming (invited paper). In *First International Conference on Scalable Information Systems (INFOSCALE 2006)*, Hong Kong, May 2006.

[20] K. Matsuzaki, Z. Hu, K. Kakehi, and M. Takeichi. Systematic derivation of tree contraction algorithms. *Parallel Processing Letters*, 15(3):321–336, 2005.

[21] K. Matsuzaki, Z. Hu, and M. Takeichi. Parallelization with tree skeletons. In *Proceedings of the 9th EuroPar Conference (EuroPar 2003)*, volume 2790 of *Lecture Notes in Computer Science*, pages 789–798, Klagenfurt, Austria, August 2003. Springer-Verlag.

[22] K. Matsuzaki, K. Kakehi, H. Iwasaki, Z. Hu, and Y. Akashi. A fusion-embedded skeleton library. In M. Danelutto, D. Laforenza, and M. Vanneschi, editors, *Proceedings of the 10th International EuroPar Conference*, volume 3149 of *Lecture Notes in Computer Science*, pages 644–653, Pisa, Italy, August/September 2004. Springer-Verlag.

[23] E. W. Mayr and R. Werchner. Optimal routing of parentheses on the hypercube. *Journal of Parallel and Distributed Computing*, 26(2):181–192, 1995.

[24] E. W. Mayr and R. Werchner. Optimal tree contraction and term matching on the hypercube and related networks. *Algorithmica*, 18(3):445–460, 1997.

[25] G. L. Miller and J. H. Reif. Parallel tree contraction and its application. In *26th Annual Symposium on Foundations of Computer Science*, pages 478–489, Portland, OR, October 1985. IEEE Computer Society Press.

[26] G. L. Miller and S.-H. Teng. Dynamic parallel complexity of computational circuits. In *Proceedings of the nineteenth annual ACM conference on Theory of computing*, pages 254–263, New York, USA, 1987. ACM Press.

[27] G. L. Miller and S.-H. Teng. Tree-based parallel algorithm design. *Algorithmica*, 19(4):369–389, 1997.

[28] G. L. Miller and S.-H. Teng. The dynamic parallel complexity of computational circuits. *SIAM Journal on Computing*, 28(5):1664–1688, 1999.

[29] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking forward. In A. B. Chaudhri, R. Unland, C. Djeraba, and W. Lindner, editors, *XML-Based Data Management and Multimedia Engineering - EDBT 2002 Workshops, EDBT 2002 Workshops XMLDM, MDDE, and YRWS, Prague, Czech Republic, March 24-28, 2002, Revised Papers*, volume 2490 of *Lecture Notes in Computer Science*, pages 109–127. Springer, 2002.

[30] D. B. Skillicorn. The Bird-Meertens formalism as a parallel model. In J. S. Kowalik and L. Grandinetti, editors, *NATO ASI Workshop on Software for Parallel Computation, NATO ARW "Software for Parallel Computation"*, volume 106 of *F*, Cetraro, Italy, June 1992. Springer-Verlag NATO ASI.

[31] D. B. Skillicorn. *Foundations of Parallel Programming.* Cambridge University Press, 1994.

[32] D. B. Skillicorn. Parallel implementation of tree skeletons. *Journal of Parallel and Distributed Computing*, 39(2):115–125, 1996.

[33] D. N. Xu, S.-C. Khoo, and Z. Hu. PType system: A featherweight parallelizability detector. In W.-N. Chin, editor, *Proceedings of Second Asian Symposium on Programming Languages and Systems (APLAS 2004)*, volume 3302 of *Lecture Note in Computer Science*, pages 197–212, Taipei, Taiwan, November 2004. Springer-Verlag.