Parallelization with Tree Skeletons

Kiminori Matsuzaki[1], Zhenjiang Hu[1,2], and Masato Takeichi[1]

[1] Graduate School of Information Science and Technology, University of Tokyo
kmatsu@ipl.t.u-tokyo.ac.jp
{hu,takeichi}@mist.i.u-tokyo.ac.jp
[2] PRESTO21, Japan Science and Technology Corporation.

Abstract. Trees are useful data structures, but to design efficient parallel programs over trees is known to be more difficult than to do over lists. Although several important tree skeletons have been proposed to simplify parallel programming on trees, few studies have been reported on how to systematically use them in solving practical problems; it is neither clear how to make a good *combination* of skeletons to solve a given problem, nor obvious even how to find suitable operators used in a single skeleton. In this paper, we report our first attempt to resolve these problems, proposing two important transformations, the *tree diffusion transformation* and the *tree context preservation transformation*. The tree diffusion transformation allows one to use familiar recursive definitions to develop his parallel programs, while the tree context preservation transformation shows how to derive associative operators that are required when using tree skeletons. We illustrate our approach by deriving an efficient parallel program for solving a nontrivial problem called the *party planning problem*, the tree version of the famous maximum-weight-sum problem.

Keywords: Parallel Skeletons, Tree Algorithms, Parallelization, Program Transformation, Algorithm Derivation.

1 Introduction

Trees are useful data types, widely used for representing hierarchical structures such as mathematical expressions or structured documents like XML. Due to irregularity (imbalance) of tree structures, developing efficient parallel programs manipulating trees is much more difficult than developing efficient parallel programs manipulating lists. Although several important tree skeletons have been proposed to simplify parallel programming on trees [4, 5, 13], few studies have been reported on how to systematically use them in solving practical problems.

Although many researchers have devoted themselves to constructing systematic parallel programming methodology using *list* skeletons [1, 2, 6, 8], few have reported the methodology with *tree* skeletons. Unlike linear structure of lists, trees do not have a linear structure, and hence the recursive functions over trees are not linear either (in the sense that there are more than one recursive call in the definition body). It is this nonlinearity that makes the parallel programming on trees complex and difficult to solve.

In this paper, we aim at a systematic method for parallel programming using tree skeletons, by proposing two important transformations, the *tree diffusion transformation* and the *tree context preservation transformation.*

– The tree diffusion transformation is an extension of the list version [8]. It shows how to decompose familiar recursive programs into equivalent parallel ones in terms of tree skeletons.
– The tree context preservation transformation is an extension of the list version [1]. It shows how to derive associative operators that are required when using tree skeletons.

In addition, to show the usefulness of these theorems, we demonstrate a derivation of an efficient parallel program for solving the *party planning problem*, using tree skeletons defined in Section 2. The party planning problem is an interesting tree version of the well-known maximum-weight-sum problem [2], which appeared as an exercise in [3].

> Professor Stewart is consulting for the president of a corporation that is planning a company party. The company has a hierarchical *tree* structure; that is, the supervisor relation forms a tree rooted at the president. The personnel office has ranked each employee with a conviviality rating, which is a real number. In order to make the party fun for all attendees, the president does not want both an employee and his or her immediate supervisor to attend. The problem is to design an algorithm making the guest list, and the goal is to maximize the sum of the conviviality rating of the guest.

It is not easy to decide which tree skeletons to use and how to combine them properly so as to solve this problem. Moreover, skeletons impose restriction (such as associativity) on the functions and operations, and it is not straightforward to find such ones.

The rest of the paper is as follows. After reviewing the tree skeletons in Section 2, we explain our two parallelization transformations for trees: the diffusion transformation in Section 3, and the context preservation transformation in Section 4. We show the experimental results in Section 5, and give conclusion in Section 6.

2 Parallel Skeletons on Trees

To simplify our presentation, we consider binary trees in this paper. The primitive parallel skeletons on binary trees are *map*, *zip*, *reduce*, *upwards accumulate* and *downwards accumulate* [13, 14], and their formal definitions using the notation of the Haskell language [9] are described in Fig 1. We will use the Haskell notation for the rest of this paper.

$$
\begin{aligned}
&\text{data } \mathit{BTree}\ \alpha\ \beta = \mathit{Leaf}\ \alpha \\
&\qquad\qquad\qquad\quad \mid\ \mathit{Node}\ (\mathit{BTree}\ \alpha\ \beta)\ \beta\ (\mathit{BTree}\ \alpha\ \beta)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{map} :: (\alpha \to \gamma, \beta \to \delta) \to \mathit{BTree}\ \alpha\ \beta \to \mathit{BTree}\ \gamma\ \delta \\
&\mathit{map}\ (f_L, f_N)\ (\mathit{Leaf}\ n) \qquad\ = \mathit{Leaf}\ (f_L\ n) \\
&\mathit{map}\ (f_L, f_N)\ (\mathit{Node}\ l\ n\ r) = \mathit{Node}\ (\mathit{map}\ (f_L, f_N)\ l)\ (f_N\ n)\ (\mathit{map}\ (f_L, f_N)\ r)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{zip} :: \mathit{BTree}\ \alpha\ \beta \to \mathit{BTree}\ \gamma\ \delta \to \mathit{BTree}\ (\alpha, \gamma)\ (\beta, \delta) \\
&\mathit{zip}\ (\mathit{Leaf}\ n)\ (\mathit{Leaf}\ n') \qquad\qquad\quad = \mathit{Leaf}\ (n, n') \\
&\mathit{zip}\ (\mathit{Node}\ l\ n\ r)\ (\mathit{Node}\ l'\ n'\ r') = \mathit{Node}\ (\mathit{zip}\ l\ l')\ (n, n')\ (\mathit{zip}\ r\ r')
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{reduce} :: (\alpha \to \gamma, \gamma \to \beta \to \gamma \to \gamma) \to \mathit{BTree}\ \alpha\ \beta \to \gamma \\
&\mathit{reduce}\ (f_L, f_N)\ (\mathit{Leaf}\ n) \qquad\ = f_L\ n \\
&\mathit{reduce}\ (f_L, f_N)\ (\mathit{Node}\ l\ n\ r) = f_N\ (\mathit{reduce}\ (f_L, f_N)\ l)\ n\ (\mathit{reduce}\ (f_L, f_N)\ r)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{uAcc} :: (\alpha \to \gamma, \gamma \to \beta \to \gamma \to \gamma) \to \mathit{BTree}\ \alpha\ \beta \to \mathit{BTree}\ \gamma\ \gamma \\
&\mathit{uAcc}\ (f_L, f_N)\ (\mathit{Leaf}\ n) \qquad\ = \mathit{Leaf}\ (f_L\ n) \\
&\mathit{uAcc}\ (f_L, f_N)\ (\mathit{Node}\ l\ n\ r) = \mathbf{let}\ l' = \mathit{uAcc}\ (f_L, f_N)\ l \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ r' = \mathit{uAcc}\ (f_L, f_N)\ r \\
&\qquad\qquad\qquad\qquad\qquad\quad\ \ \mathbf{in}\ \mathit{Node}\ l'\ (f_N\ (\mathit{root}\ l')\ n\ (\mathit{root}\ r'))\ r'
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{dAcc} :: (\gamma \to \gamma \to \gamma) \to (\beta \to \gamma, \beta \to \gamma) \to \mathit{BTree}\ \alpha\ \beta \to \gamma \to \mathit{BTree}\ \gamma\ \gamma \\
&\mathit{dAcc}\ (\oplus)\ (f_L, f_R)\ (\mathit{Leaf}\ n)\ c \qquad\ = \mathit{Leaf}\ c \\
&\mathit{dAcc}\ (\oplus)\ (f_L, f_R)\ (\mathit{Node}\ l\ n\ r)\ c = \mathit{Node}\ (\mathit{dAcc}\ (\oplus)\ (f_L, f_R)\ l\ (c \oplus f_L\ n))\ c \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (\mathit{dAcc}\ (\oplus)\ (f_L, f_R)\ r\ (c \oplus f_R\ n))
\end{aligned}
$$

Fig. 1. Definitions of five primitive skeletons

The *map* skeleton $\mathit{map}\ (f_L, f_N)$ applies function f_L to each leaf and function f_N to each internal node. The *zip* skeleton accepts two trees of the same shape and returns a tree whose nodes are pairs of corresponding two nodes of the original two trees. The *reduce* skeleton $\mathit{reduce}\ (f_L, f_N)$ reduces a tree into a value by applying f_L to each leaf, and f_N to each internal node upwards. Similar to *reduce*, the *upwards accumulate* skeleton $\mathit{uAcc}\ (f_L, f_N)$ applies f_L to each leaf and f_N to each internal node in a bottom-up manner, and returns a tree of the same shape as the original tree. The *downwards accumulate* skeleton $\mathit{dAcc}\ (\oplus)\ (f_L, f_R)\ c$ computes by propagating accumulation parameter c downwards, and the accumulation parameter is updated by $\oplus$ and f_L when propagated to left child, or updated by $\oplus$ and f_R when propagated to right child.

To guarantee the existence of efficient implementation for the parallel skeletons, we have requirement on the operators and functions used in the above skeletons.

Definition 1 (Semi-Associative). A binary operator $\otimes$ is said to be *semi-associative* if there is an associative operator $\oplus$ such that for any a, b, c, $(a \otimes b) \otimes c = a \otimes (b \oplus c)$. □

Definition 2 (Quasi-Associative). A binary operator $\oplus$ is said to be quasi-associative if there is a semi-associative operator $\otimes$ and a function f such that for any a, b, $a \oplus b = a \otimes f\ b$. □

Definition 3 (Bi-Quasi-Associative). A ternary operator f is said to be *bi-quasi-associative* if there is a semi-associative operator $\otimes$ and two functions f'_L, f'_R such that for any l, n, r, $f\ l\ n\ r = l \otimes f'_L\ n\ r = r \otimes f'_R\ n\ l$. We can fix a bi-quasi-associative operator f by providing $\otimes$, $\oplus$ (associative operator for $\otimes$), f'_L and f'_R, therefore, we will write f with 4-tuple as $f \equiv [[\otimes, \oplus, f'_L, f'_R]]$. $\square$

Based on the tree contraction technique [12], we require the f_N used in the *reduce* and *upwards accumulate* be bi-quasi-associative, and $\oplus$ in *downwards accumulate* be associative. We omit the detailed description of the cost for each skeleton. Informally, if all the operators used in the skeletons use constant time, all skeletons can be implemented in at most $O(\log N)$ parallel time using enough processors, where N denotes the number of nodes in the tree.

3 Tree Diffusion Theorem

Hu et al. proposed the diffusion theorem (on lists) [8], with which we can directly derive efficient combinations of skeletons from recursive programs. In this section, we start by formalizing a very general tree diffusion theorem, then discuss three practical cases, and finally derive a combination of skeletons for the party planning problem.

Theorem 1 (Tree Diffusion). Let f be defined in the following recursive way over binary trees:

$$\begin{aligned} &f\ (\mathit{Leaf}\ n)\ c &&= g_L\ \ (n, c) \\ &f\ (\mathit{Node}\ l\ n\ r)\ c &&= g_N\ (f\ l\ (c \otimes h_L\ n))\ (n, c)\ (f\ r\ (c \otimes h_R\ n)) \end{aligned}$$

where g_N is a bi-quasi-associative operator, $\otimes$ is an associative operator, and g_L, h_L, h_R are user-defined functions. Then f can be equivalently defined in terms of the tree skeletons as follows.

$$\begin{aligned} f\ xt\ c = &\ \mathbf{let}\ ct = \mathit{dAcc}\ (\otimes)\ (h_L, h_R)\ xt\ c \\ &\ \mathbf{in}\ \mathit{reduce}\ (g_L, g_N)\ (\mathit{zip}\ xt\ ct) \end{aligned}$$

Proof Sketch: This can be proved by induction on the structure of xt. Due to the limitation of space, the proof is given in the technical report [11]. $\square$

This theorem is very general. Practically, It is often the case that the function f returns a tree with the same shape as the input. If we naively apply this diffusion theorem, we will have a costly *reduce* skeleton for combining all sub-trees. To remedy this situation, we propose the following two useful specializations, in which we use appropriate skeletons rather than *reduce*.

The first specialization deals with the function whose computation of new values for each node depends on the original value and the accumulation parameter. For each internal node, such a function f can be defined as $f\ (\mathit{Node}\ l\ n\ r) = \mathit{Node}\ (f\ l\ (c \otimes h_L\ n))\ (g_N\ (n, c))\ (f\ r\ (c \otimes h_R\ n))$, and this function can be efficiently computed by *map* rather than *reduce*.

```
$ppp\ xt = ppp'\ xt\ True$

$ppp'\ (Leaf\ n)\ c\qquad = Leaf\ c$
$ppp'\ (Node\ l\ n\ r)\ c = \mathbf{let}\ (l_m, l_u) = mis\ l$
                              $(r_m, r_u) = mis\ r$
                          $\mathbf{in}\ Node\ (ppp'\ l\ (\text{if } c \text{ then } False \text{ else } (l_m > l_u)))\ c$
                                 $(ppp'\ r\ (\text{if } c \text{ then } False \text{ else } (r_m > r_u)))$
$mis\ (Leaf\ n)\qquad = (n, 0)$
$mis\ (Node\ l\ n\ r) = \mathbf{let}\ (l_m, l_u) = mis\ l$
                           $(r_m, r_u) = mis\ r$
                       $\mathbf{in}\ (l_u + n + r_u, (l_m \uparrow l_u) + (r_m \uparrow r_u))$
```

Fig. 2. A sequential program for party planning program

The second specialization deals with the function whose computation of new values for each node depends on the original value, the accumulation parameter and the new value of its children. For each internal node, such a function f can be defined as $f\ (Node\ l\ n\ r)\ c = Node\ l'\ (g_N\ (root\ l')\ (n, c)\ (root\ r'))\ r'$ where $l' = f\ l\ (c \otimes h_L\ n)$ and $r' = f\ l\ (c \otimes h_R\ n)$. This function can be efficiently computed by *upwards accumulate* rather than *reduce*.

Let us discuss another practical matter for the case where the function f calls an auxiliary function k to compute over the sub-trees. Such a function can be defined as follows.

$$
\begin{aligned}
&f\ (Leaf\ n)\ c &&= Leaf\ (g_L\ ((\_, n, \_), c))\\
&f\ (Node\ n\ l\ r)\ c &&= \mathbf{let}\ n' = (k\ l, n, k\ r)\\
& &&\quad\ \mathbf{in}\ Node\ (f\ l\ (c \otimes h_L\ n'))\ (g_N\ (n', c))\ (f\ r\ (c \otimes h_R\ n'))\\
&k\ (Leaf\ n) &&= k_L\ n\\
&k\ (Node\ l\ n\ r) &&= k_N\ (k\ l)\ n\ (k\ r)
\end{aligned}
$$

It is a little difficult to efficiently parallelize this recursive function into the combination of primitive skeletons, because there are multiple traversals over the trees, and naive computation of f will make redundant function calls of k. By making use of the tupling transformation and the fusion transformation [7], we can parallelize the function efficiently. In the following, we use a function *gather_ch*, which accepts two trees of the same shape and makes a triple for each node. The triple consists of a node of the first tree and two immediate children of the second tree. Detailed discussions are referred to [11].

Corollary 1 (Paramorphic Diffusion). The function f defined above can be diffused into the following combination of skeletons if k_N is a bi-quasi-associative operator, and $\otimes$ is associative.

$$
\begin{aligned}
f\ xt\ c = \mathbf{let}\ &yt = gather\_ch\ xt\ (uAcc\ (k_L, k_N)\ xt)\\
\mathbf{in}\ &dAcc\ (\otimes)\ (h_L, h_R)\ yt\ c
\end{aligned}
\qquad \square
$$

Having shown the diffusion theorem and its corollaries, we now try to derive a parallel program for the party planning problem. By making use of dynamic programming technique, we can obtain an efficient sequential program as shown in Fig 2. Here, the function *mis* accepts a tree, and returns a pair of values which are the maximum independent sums when the root of the input is $\underline{m}$arked or $\underline{u}$nmarked. The recursive function ppp' is defined with an accumulation parameter, which represents a node to be marked or unmarked. The recursive function ppp' is a paramorphic function because it calls an auxiliary function *mis* on each sub-tree, therefore, let us use paramorphic diffusion theorem to derive the following program in terms of skeletons.

$$\begin{aligned} &ppp\ xt = ppp'\ xt\ True \\ &ppp'\ xt\ c = \textbf{let}\ yt = gather\_ch\ xt\ (uAcc\ (mis_L, \underline{mis_N})\ xt) \\ &\qquad\qquad\ \textbf{in}\ dAcc\ \underline{(\otimes)\ (h_L, h_R)}\ yt\ c \end{aligned}$$

However, we have not yet parallelized the underlined parts successfully. First, from the definition of the sequential program, we can derive $mis_L\ n = (n, 0)$ and $mis_N\ (l_m, l_u)\ n\ (r_m, u_u) = (l_u + n + r_u, (l_m \uparrow l_u) + (r_m \uparrow r_u))$, however, we have still to show the bi-quasi-associativity of mis_N. Second, we have to derive an associative operator $\otimes$ and two functions h_L and h_R such that $c \otimes h_L\ ((l_m, l_u), n, (r_m, r_u)) =$ if c then *False* else $(l_m > l_u)$ and almost the same equation for h_R hold. In the following section, we will see how to derive those operators.

4 Tree Context Preservation

The parallel skeletons require the operators used in them to be (bi-quasi)-associative, however, it is not straightforward to find such ones for many practical problems. For linear self-recursive programs, Chin et al. proposed the *context preservation transformation* [1], with which one can systematically derive such operators based on the associativity of function composition. In this section, we will extend the transformation theorem for tree skeletons. Our main idea is to resolve the non-linear functions over trees into two linear recursive functions, and then we can consider the context preservation on these two linear functions. We start by introducing the basic notations and concepts about contexts.

Definition 4 (Context Extraction [1]). Given an expression E and sub-terms $\langle e_1, \ldots, e_n\rangle$, we shall express its extraction by: $E \Longrightarrow E'\langle e_1, \ldots, e_n\rangle$. The context E' has a form of $\lambda\langle\_\__1, \ldots, \_\__n\rangle.[e_i \mapsto \_\__i]_{i=1}^{n} E$, where $\_\__i$ denotes a new hole and $[e_i \mapsto \_\__i]_{i=1}^{n} E$ denotes a substitution notation of e_i in E to $\_\__i$. □

Definition 5 (Skeletal Context [1]). A context E is said to be a *skeletal context* if every sub-term in E contains at least one hole. Given a context E, we can make it into a skeletal one E_S by extracting all sub-terms that do not contain holes. This process shall be denoted by $E \Longrightarrow_S E_S\langle e_i\rangle_{i \in N}$ □

Definition 6 (Context Transformation [1]). A context may be transformed (or simplified) by either applying laws or unfolding. We shall denote this process as $E \Longrightarrow_T E'$. □

Definition 7 (Context Preservation Modulo Replication [1]). A context E with one hole is said to be preserved modulo replication if there is a skeletal context E_S, $E \Longrightarrow_S E_S\langle t_i \rangle$ and $E_S\langle \alpha_i \rangle \circ E_S\langle \beta_i \rangle = E_S\langle \gamma_i \rangle$ hold, where α_i and β_i are variables, and γ_i are sub-terms without holes. $\square$

Now, we will discuss about the functions which can be transformed into a program with *uAcc*.

Definition 8 (Simple Upwards Recursive Function). A function is said to be a *simple upwards recursive function* (*SUR-function* for short) if it has the following form.

$$\begin{aligned} f\ (\mathit{Leaf}\ n) &= f_L\ n \\ f\ (\mathit{Node}\ l\ n\ r) &= f_N\ (f\ l)\ n\ (f\ r) \end{aligned} \quad \square$$

The inductive case of an SUR-function has two recursive calls, $f\ l$ and $f\ r$, therefore, we cannot apply the Chin's theorem. To resolve this non-linearity, we define the extraction of two linear recurring contexts from an SUR-function, and extended context preservation for these two contexts as shown in the following.

Definition 9 (Left(Right)-Recurring Context). For the inductive case of an SUR-function, we can extract the *left(right)-recurring context* E^L (E^R) by abstracting either of the recurring terms: $f\ (\mathit{Node}\ l\ n\ r) = E^L\langle f\ l \rangle = E^R\langle f\ r \rangle$. $\square$

Definition 10 (Mutually Preserved Contexts). Two linear recurring contexts E^L, E^R are said to be *mutually preserved* if there exists a skeletal context E_S such that $E^L \Longrightarrow_S E_S\langle g^l\ n\ r \rangle$, $E^R \Longrightarrow_S E_S\langle g^r\ n\ l \rangle$ and $E_S\langle \alpha \rangle \circ E_S\langle \beta \rangle = E_S\langle \gamma \rangle$ hold. Here, γ is a sub-terms computed only with variables α and β. $\square$

Based on the idea of tree contraction algorithm, we can parallelize the SUR-function as shown in the following theorem. Due to the limitation of space we omit the proof, which is given in the technical report [11].

Theorem 2 (Context Preservation for SUR-function). The SUR-function function f can be parallelized to $f = \mathit{uAcc}\ (f_L, f_N)$ if there exist a skeletal context E_S such that $E^L \Longrightarrow_S E_S\langle g^l\ n\ r \rangle$, $E^R \Longrightarrow_S E_S\langle g^r\ n\ l \rangle$ and $E_S\langle \alpha \rangle \circ E_S\langle \beta \rangle = E_S\langle \gamma \rangle$ hold. Here, f_N is a bi-quasi-associative operator such as $f_N \equiv [[\oplus, \otimes, g^l, g^r]]$ where $x \oplus \alpha = E_S\langle \alpha \rangle \langle x \rangle$ and $\beta \otimes \alpha = \gamma$. $\square$

Next, we discuss about the functions which can be transformed into a program with *dAcc*. As is the case of *upwards accumulate*, based on the tree contraction algorithm, we can parallelize a non-linear function by extracting two linear contexts and showing these contexts to be mutually preserved. Due to the limitation of space, we only show the definitions and theorem for this.

Definition 11 (Simple Downwards Recursive Function). A function is said to be a *simple downwards recursive function* (*SDR-function* for short) if it has the following form.

$$\begin{aligned} f\ (\mathit{Leaf}\ n)\ c &= \mathit{Leaf}\ c \\ f\ (\mathit{Node}\ l\ n\ r)\ c &= \mathit{Node}\ (f\ l\ (f_L\ c\ n))\ c\ (f\ r\ (f_R\ c\ n)) \end{aligned} \quad \square$$

Definition 12 (Recurring Contexts for SDR-function). For the inductive case of an SDR-function f, we can obtain two *recurring contexts* D^L, D^R by abstracting the recursive calls on the accumulative parameter respectively, $f\ (Node\ l\ n\ r)\ c = Node\ (f\ l\ D^L\langle c\rangle)\ c\ (f\ r\ D^R\langle c\rangle)$. □

Theorem 3 (Context Preservation for SDR-function). The SDR-function f can be parallelized to $f\ xt\ c = map\ ((c\otimes),(c\otimes))\ (dAcc\ (\oplus)\ (g^l, g^r)\ \iota_\oplus)$ if there exist a skeletal context E_S such that $D^L \Longrightarrow_S D_S\langle g^l\ n\rangle$, $D^R \Longrightarrow_S D_S\langle g^r\ n\rangle$ and $D_S\langle\alpha\rangle \circ D_S\langle\beta\rangle = D_S\langle\gamma\rangle$ hold. Here, the operators are defined as $\beta \oplus \alpha = \gamma$ and $c \otimes \alpha = D_S\langle\alpha\rangle\langle c\rangle$, and $\iota_\oplus$ is the unit of $\oplus$. □

Having shown the context preservation theorems for trees, we now demonstrate how these theorems work by deriving an associative operator $\otimes$ and functions h_R, h_L in the diffused program in Section 3. The corresponding part is defined recursively as follows.

$$\begin{aligned}
&ppp'\ (Node\ l\ ((l_m, l_u), n, (r_m, r_u))\ r)\ c \\
&\quad = Node\ (ppp'\ l\ (\text{if } c \text{ then } False \text{ else } (l_m > l_u)))\ c \\
&\qquad\quad (ppp'\ r\ (\text{if } c \text{ then } False \text{ else } (r_m > r_u)))
\end{aligned}$$

From this definition, we can obtain the following two linear recurring contexts by abstracting recursive calls.

$$\begin{aligned}
D^L &= \lambda\langle c\rangle.\text{if } c \text{ then } False \text{ else } (l_m > l_u) \\
D^R &= \lambda\langle c\rangle.\text{if } c \text{ then } False \text{ else } (r_m > r_u)
\end{aligned}$$

We can show that these two contexts are mutually recursive because the skeletal context $D_S = \lambda\langle \_\__1, \_\__2\rangle.\lambda\langle c\rangle.\text{if } c \text{ then } \_\__1 \text{ else } \_\__2$ satisfies our requirement.

$$\begin{aligned}
&D^L = D_S\langle g^l\ ((l_m, l_u), n, (r_m, r_u))\rangle, \quad D^R = D_S\langle g^r\ ((l_m, l_u), n, (r_m, r_u))\rangle \\
&\textbf{where } g^l\ ((l_m, l_u), n, (r_m, r_u)) = (false, (l_m > l_u)) \\
&\qquad\quad g^r\ ((l_m, l_u), n, (r_m, r_u)) = (false, (r_m > r_u))
\end{aligned}$$

$$\begin{aligned}
&D_S\langle\alpha_1, \alpha_2\rangle \circ D_S\langle\beta_1, \beta_2\rangle \\
&= \lambda\langle c\rangle.\text{if } c \text{ then } (\text{if } \beta_1 \text{ then } \alpha_1\ else\ \alpha_2)\ else\ (\text{if } \beta_2 \text{ then } \alpha_1 \text{ else } \alpha_2) \\
&= D_S\langle \text{if } \beta_1 \text{ then } \alpha_1 \text{ else } \alpha_2, \text{if } \beta_2 \text{ then } \alpha_1 \text{ else } \alpha_2\rangle
\end{aligned}$$

From the derivations above, we can apply theorem 3 to obtain an efficient parallel program with *map* and *downwards accumulate*. The whole parallel program for the party planning problem is shown in Fig 3. Detailed derivations are referred to [11].

5 An Experiment

We have conducted an experiment on the party planning problem. We have coded our algorithm using C++, the MPI library and our implementation of tree skeletons [10]. We have used a tree of 999,999 nodes for our experiment.

Fig 4 shows the result of the program executed on our PC-Cluster using 1 to 12 processors. This result is shown in the speedup excluding partitioning and flattening of the tree. The almost linear speedup shows the effectiveness of the program derived by our theorems.

```
$ppp\ xt = \mathbf{let}\ yt = gather\_ch\ xt\ (uAcc\ (mis_L, mis_N)\ xt)$
          $\mathbf{in}\ map\ (fst, fst)\ (dAcc\ (\odot)\ (h_L, h_R)\ yt\ \iota_\odot)$
$\mathbf{where}$
$mis_L = (n, 0)$
$mis_N \equiv [[\oplus, \otimes, f^L, f^R]]$
$(\beta_1, \beta_2, \beta_3, \beta_4) \oplus (\alpha_1, \alpha_2, \alpha_3, \alpha_4) = ((\beta_1 + \alpha_1) \uparrow (\beta_3 + \alpha_2),$
  $(\beta_2 + \alpha_1) \uparrow (\beta_4 + \alpha_2),\ (\beta_1 + \alpha_3) \uparrow (\beta_3 + \alpha_4),\ (\beta_2 + \alpha_3) \uparrow (\beta_4 + \alpha_4))$
$(x_m, x_u) \otimes (\alpha_1, \alpha_2, \alpha_3, \alpha_4) = ((x_m + \alpha_1) \uparrow (x_u + \alpha_2),\ (x_m + \alpha_3) \uparrow (x_u + \alpha_4))$
$f^L\ n\ (r_m, r_u) = (-\infty,\ n + r_u,\ r_m \uparrow r_u,\ r_m \uparrow r_u)$
$f^R\ n\ (l_m, l_u) = (-\infty,\ n + l_u,\ l_m \uparrow l_u,\ l_m \uparrow l_u)$

$(\beta_1, \beta_2) \odot (\alpha_1, \alpha_2) = (\text{if } \beta_1 \text{ then } \alpha_1 \text{ else } \alpha_2,\ \text{if } \beta_2 \text{ then } \alpha_1 \text{ else } \alpha_2)$
$\iota_\odot = (True,\ False)$
$h_L\ ((l_m, l_u), n, (r_m, r_u)) = (False,\ (l_m > l_u))$
$h_R\ ((l_m, l_u), n, (r_m, r_u)) = (False,\ (r_m > r_u))$
```

Fig. 3. Parallel program for party planning problem

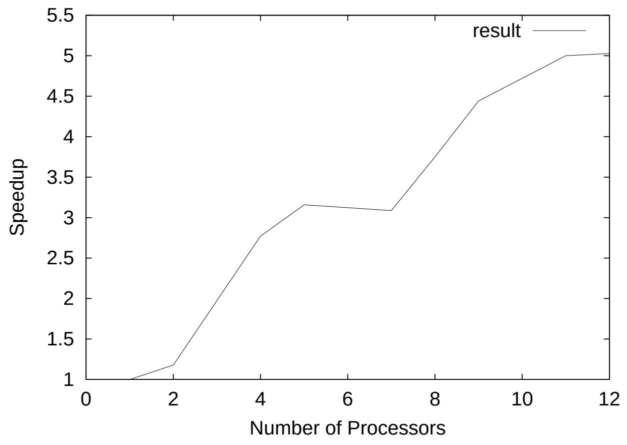

Fig. 4. Experiment result

6 Conclusion

In this paper, we have proposed two parallelization transformations, the *tree diffusion transformation* and the *context preservation transformation*, for helping programmers to systematically derive efficient parallel programs in terms of tree skeletons from the recursive programs. The list versions of these two theorems have been proposed and shown important in skeletal parallel programming, which once in fact motivated us to see if we could generalize them for trees. Due to the non-linearity of the tree structures, it turns out to be more difficult than we had expected. Although the usefulness of our theorems await more evidence, our successful derivation of the *first* skeletal parallel program for solving the party planning problem and the good experiment result have indicated that this is a good start and is worth further investigation.

We are currently working on generalizing the context preservation theorem so that we can relax conditions of the skeletons. In addition, we are figuring out whether we can automatically parallelize the recursive programs on trees.

References

1. 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.
2. M. Cole. Parallel programming, list homomorphisms and the maximum segment sum problems. Report CSR-25-93, Department of Computing Science, The University of Edinburgh, May 1993.
3. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, second edition, 2001.
4. J. Gibbons. *Algebras for Tree Algorithms.* PhD thesis, Programming Research Group, Oxford University, 1991. Available as Technical Monograph PRG-94.
5. J. Gibbons. Computing downwards accumulations on trees quickly. In G. Gupta, G. Mohay, and R. Topor, editors, *Proceedings of 16th Australian Computer Science Conference*, volume 15 (1), pages 685–691. Australian Computer Science Communications, February 1993.
6. 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.
7. Z. Hu, H. Iwasaki, and M. Takeichi. Construction of list homomorphisms by tupling and fusion. In *21st International Symposium on Mathematical Foundation of Computer Science, LNCS 1113*, pages 407–418, Cracow, September 1996. Springer-Verlag.
8. 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.
9. S. Peyton Jones and J. Hughes, editors. *Haskell 98: A Non-strict, Purely Functional Language.* Available online: `http://www.haskell.org`, February 1999.
10. K. Matsuzaki, Z. Hu, and M. Takeichi. Implementation of parallel tree skeletons on distributed systems. In *Proceedings of The Third Asian Workshop on Programming Languages And Systems*, pages 258–271, Shanghai, China, 2002.
11. K. Matsuzaki, Z. Hu, and M. Takeichi. Parallelization with tree skeletons. Technical Report METR 03-21, Mathematical Informatics, Graduate School of Information Science and Technology, University of Tokyo, 2003.
12. M. Reid-Miller, G. L. Miller, and F. Modugno. List ranking and parallel tree contraction. In John H. Reif, editor, *Synthesis of Parallel Algorithms*, chapter 3, pages 115–194. Morgan Kaufmann Publishers, 1996.
13. D. B. Skillicorn. *Foundations of Parallel Programming.* Cambridge University Press, 1994.
14. D. B. Skillicorn. Parallel implementation of tree skeletons. *Journal of Parallel and Distributed Computing*, 39(2):115–125, 1996.