Implementation of Parallel Tree Skeletons on Distributed Systems

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

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

Abstract. Trees are useful data types, but developing efficient parallel programs manipulating trees is known to be difficult, because of their irregular and imbalance structure. Parallel tree skeletons are designed to ease parallel programming by encouraging programmers to build parallel programs by combining them. However, for distributed systems, efficient implementations of these parallel tree skeletons are known to be hard.
In this paper, we propose an implementation of parallel tree skeletons that run efficiently on distributed systems. Our approach is as follows, first we partition a tree by using m-bridge technique, and locally compute by composing functions, then propagate the results over the tree. The results of several experiments shows that our approach is promising, even if the tree is imbalanced. Furthermore, we present the conditions for efficient implementation. ...

1 Introduction

In late years, distributed systems like PC-clusters are widely available, but developing efficient parallel programs is still known to be difficult because programmers must be aware of arrangement of resource, communication between processors, synchronization and etc. To solve this problem, *skeletal parallel programming* [3] has been proposed to encourage programmers to build parallel programs with ready-made skeletons. By using efficiently implemented skeletons, programmers can build efficient parallel programs without the deep knowledge of parallel algorithms and hardware. There are many studies about parallel skeletons for lists [1, 8] and nested list [10].

Trees are useful data types, but developing efficient parallel programs manipulating trees is known to be difficult, because of their irregular and imbalanced structure. Parallel tree skeletons [9] are designed to ease parallel programming by encouraging programmers to build parallel programs by combining them. However, for distributed systems, efficient implementations of these parallel tree skeletons are known to be hard.

In this paper, we propose an implementation of parallel tree skeletons, by first partitioning tree into connected subtrees using the m-bridge technique [7], and then locally computing in parallel using function composition, and finally

propagating the results over the tree. The advantages of our implementation are as follows.

- By using the m-bridge technique to partition a binary tree, we can obtain subtrees with almost the equal number of nodes even if the original tree is imbalanced. The partitioned subtrees also form a binary tree, and this is important for the applicability of our efficient implementation.
- By the function composition technique [4], we can independently compute each subtree that has unknown values, and parallel computation cost of this local evaluation is $O(n/p)$ where n is the number of nodes of the original tree and p is the number of processors.
- We have made several experiments and the results show our implementation approach is practically efficient.

The organization of this paper is as follows. In Section 2, we review five primitive tree skeletons and propose our diffusion theorem with an example. We explain our implementation approach and show distribution form of parallel tree skeletons in Section 3, and describe the detail of implementation and the results of experiments in Section 4. We formalize the conditions for efficient implementation in Section 5, and we conclude and discuss our future work in Section 6.

2 Parallel Skeletons on Trees

The primitive parallel skeletons on binary trees are map, zip, reduce, upwards accumulate and downwards accumulate [8, 9], and their formal definitions using the notation of Haskell language is described in Fig. 1.

- *map* (f_L, f_N) applies function f_L to each leaf and function f_N to each internal node.
- *zip* recieves two trees of the same shape and returns a tree whose nodes are pairs of values of the original two trees.
- *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 in a bottom-up manner.
- Similar to reduce, *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.
- *dAcc* (f_L, f_R) computes by propagating accumulation parameter c in a top-down manner. Accumulation parameters are updated by f_L when they are propagated to left child, or updated by f_R when they are propagated to right child.

With these primitive parallel tree skeletons, programmers can build a parallel program as a combination of them. But, how to efficiently combine them remains as a difficult problem. The *Diffusion Theorem* [6] enables us to derive effecient skeleton parallel programs from recursive programs systematically.

$$
\begin{array}{l}
BTree\ \alpha\ \beta = Leaf\ \alpha \\
\qquad\qquad \mid\ Node\ \beta\ (BTree\ \alpha\ \beta)\ (BTree\ \alpha\ \beta) \\
\\
map :: (\alpha \to \gamma, \beta \to \delta) \to BTree\ \alpha\ \beta \to BTree\ \gamma\ \delta \\
map\ (f_L, f_N)\ (Leaf\ n) \qquad = Leaf\ (f_L\ n) \\
map\ (f_L, f_N)\ (Node\ n\ l\ r) = Node\ (f_N\ n)\ (map\ (f_L, f_N)\ l)\ (map\ (f_L, f_N)\ r) \\
\\
zip :: BTree\ \alpha\ \beta \to BTree\ \gamma\ \delta \to BTree\ (\alpha, \gamma)\ (\beta, \delta) \\
zip\ (Leaf\ n)\ (Leaf\ n') \qquad\qquad = Leaf\ (n, n') \\
zip\ (Node\ n\ l\ r)\ (Node\ n'\ l'\ r') = Node\ (n, n')\ (zip\ l\ l')\ (zip\ r\ r') \\
\\
reduce :: (\alpha \to \gamma, \beta \to \gamma \to \gamma \to \gamma) \to BTree\ \alpha\ \beta \to \gamma \\
reduce\ (f_L, f_N)\ (Leaf\ n) \qquad = f_L\ n \\
reduce\ (f_L, f_N)\ (Node\ n\ l\ r) = f_N\ n\ (reduce\ (f_L, f_N)\ l)\ (reduce\ (f_L, f_N)\ r) \\
\\
uAcc :: (\alpha \to \gamma, \beta \to \gamma \to \gamma \to \gamma) \to BTree\ \alpha\ \beta \to BTree\ \gamma\ \gamma \\
uAcc\ (f_L, f_N)\ (Leaf\ n) \qquad = Leaf\ (f_L n) \\
uAcc\ (f_L, f_N)\ (Node\ n\ l\ r) = Node\ (f_N)\ n\ (root\ lt)\ (root\ rt))\ lt\ rt \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{where}\ lt\ = uAcc\ (f_L, f_N)\ l \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad rt = uAcc\ (f_L, f_N)\ r \\
\\
dAcc :: (\beta \to \gamma \to \gamma, \beta \to \gamma \to \gamma) \to BTree\ \alpha\ \beta \to BTree\ \gamma\ \gamma \\
dAcc\ (f_L, f_R)\ (Leaf\ n)\ c \qquad = Leaf\ c \\
dAcc\ (f_L, f_R)\ (Node\ n\ l\ r)\ c = Node\ c\ (dAcc\ (f_L, f_R)\ l\ (f_L\ n\ c)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (dAcc\ (f_L, f_R)\ r\ (f_R\ n\ c))
\end{array}
$$

Fig. 1. The definitions of five primitive tree skeletons

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

$$
\begin{array}{l}
f\ (Leaf\ n)\ c \quad = Leaf\ \ (g_L\ (n, c)) \\
f\ (Node\ n\ l\ r) = Node\ (g_N\ \ (n, c)) \\
\qquad\qquad\qquad\qquad\quad (f\ l\ (h_L\ n\ c)) \\
\qquad\qquad\qquad\qquad\quad (f\ r\ (h_R\ n\ c))
\end{array}
$$

where g_L, g_R, h_L and h_R are given functions. Then f can be diffused to the following form

$$
\begin{array}{l}
f\ t\ c = \textbf{let}\ ct = dAcc\ (h_L, h_R)\ t\ c \\
\qquad\quad \textbf{in}\ map\ (g_L, g_N)\ (zip\ t\ ct).
\end{array}
$$

□

To illustrate how the diffusion theorem works, consider the problem of in-order numbering of nodes on a binary tree. A recursive program for solving this problem is:

$$
\begin{array}{l}
ion\ (Leaf\ n)\ c \qquad = Leaf\ \ c \\
ion\ (Node\ n\ l\ r)\ c = Node\ (c + size\ l) \\
\qquad\qquad\qquad\qquad\quad (ion\ l\ c) \\
\qquad\qquad\qquad\qquad\quad (ion\ r\ (c + size\ l + 1))
\end{array}
$$

where

$$\begin{aligned} size\ (Leaf\ n) &= 1 \\ size\ (Node\ n\ l\ r) &= 1 + size\ l + size\ r. \end{aligned}$$

By calculating $size\ l$ for each node in advance, we can apply the diffusion theorem to obtain the following skeletal parallel program:

$$\begin{aligned} ion'\ t\ c = \textbf{let}\ & st = lchild\ (uAcc\ (id, sum3)\ (map\ (\lambda x.1, \lambda x.1)\ t)) \\ & ct = dAcc\ (h_L, h_N)\ st\ c \\ \textbf{in}\ & map\ (g_L, g_R)\ (zip\ st\ ct) \end{aligned}$$

where

$$\begin{aligned} lchild\ (Node\ n\ l\ r) &= root\ l \\ sum3\ n\ l\ r &= n + l + r \\ g_1\ (s, c) &= c \\ g_2\ (s, c) &= s + c \\ h_1\ s\ c &= c \\ h_2\ s\ c &= s + 1 + c. \end{aligned}$$

If each of the skeleton used in the above program can be efficiently implemented in at most $O(\log n)$ parallel time, we can obtain an efficient parallel program whose parallel computation cost is $O(\log n)$.

3 Distributive Form of Parallel Tree Skeletons

When we build a parallel program that runs efficiently on distributed systems, we must be aware of the distribution of data and communication between processors. In generally, under the message passing programming model, programs runs communicating with other processors. Our approach to implementing the tree skeletons consists of five steps:

1. Parition of a tree
2. Local computation by function composition
3. Global propagation
4. Local computation by substitution
5. Flattening of the subtree

In the following, we briefly review each step before giving the formal distributive form for each tree skeleton.

Partition of a tree On distributed systems, the communication cost is so expensive that how to partition a tree is an important problem. In our implementation, we use the m-bridge technique to partition a tree, which will be explained in Section 4.1, and this method enjoys nice property of load-balancing and good shape of partitioned subtrees. The type of *partition* can be represented as

$$partition :: BTree\ \alpha\ \beta \to BTree\ (\alpha \to \alpha \to BTree\ \alpha\ \beta)$$

where $\alpha \to \alpha \to BTree\ \alpha\ \beta$ stands for a tree containing two holes for connection to other subtrees, and if the holes are filled with values of leaf type then the value of the subtree can be obtained.

We define three operations, *root*, *left* and *right*, which pick up distinguished nodes from a partitioned subtree.

$$\begin{aligned} root &:: (\alpha \to \alpha \to BTree\ \alpha\ \beta) \to \beta \\ left &:: (\alpha \to \alpha \to BTree\ \alpha\ \beta) \to \beta \\ right &:: (\alpha \to \alpha \to BTree\ \alpha\ \beta) \to \beta \end{aligned}$$

The function *root* picks up the root node from a subtree, *left* picks up the parent node of the left hole, and *right* picks up the parent node of the right hole.

Local computation by function composition In this step, we compute on each subtree independently. The difficulty is that at this point the results on other processors are unknown. Our idea is by applying the function composition technique to trees, which has been proposed on lists [4], to represent the result of each node by function and to compute this function for each node locally. After we obtain the results from other processors, we can compute the results of the subtree immediately.

For example, in the case of reduce, each subtree has two holes, and the result of local reduction becomes a function with two arguments, and local reduction $local_r$ has the type of

$$local_r :: (\alpha \to \alpha \to BTree\ \alpha\ \beta) \to (\gamma \to \gamma \to \gamma).$$

However, how to derive this $local_r$ function from given function is a difficult problem and it will be explained in Section 5.

The parallel computation cost in this step is $O(k)$ where k represents the number of nodes in the partitioned subtree.

Global propagation After local computation, we globally propagate values to fill holes of each subtree. We obtain the results of local composed functions in the previous step, so we compute this global propagation by substituting the values for the arguments of functions on other subtrees.

The communication cost of this step is $O(p)$ in sequential way where p represents the number of processors. However, the global shape of partitioned subtrees is also a binary tree, so we can use the tree contraction technique to decrease the communication cost to $O(\log p)$.

Local computation by substitution After global data propagation, we again compute on each subtree locally by substituting the results for the arguments the function of each node obtained in the second step. We can compute independently in this step, and the parallel computation cost is $O(k)$ where k represents the number of nodes in the partitioned subtree.

Flattening of the subtrees Finally, we apply *flatten*, which is the inverse operation of *partition*, to flatten the subtrees to the original tree. Function *flatten* has following type:

$$flatten :: BTree\ (\alpha \to \alpha \to BTree\ \alpha\ \beta) \to BTree\ \alpha\ \beta.$$

In the following, we show how the five skeletons can be described in our distributed form.

Distributive form of map

Skeleton *map* applies either of paired functions to each node and these functions need no informations from others, so we can implement *map* on distributed systems by applying *map* to each partitioned subtree. The distributed form of *map* is:

$$\begin{aligned} &map :: (\alpha \to \gamma, \beta \to \delta) \to BTree\ \alpha\ \beta \to BTree\ \gamma\ \delta \\ &map\ f = flatten \circ map\ (local_m\ f) \circ partition \end{aligned}$$

where $local_m\ f$ has the type of

$$local_m\ f :: (\alpha \to \alpha \to BTree\ \alpha\ \beta) \to (\gamma \to \gamma \to BTree\ \gamma\ \delta).$$

It applies f to each node in the partitioned subtree. If there are no holes on a subtree, $local_m$ is the same as *map*.

Distributive form of zip

Similar to *map*, *zip* also needs no informations from other nodes, so we can implement *zip* on distributed systems by applying *zip* to each partitioned subtree. The distributed form of *zip* is:

$$\begin{aligned} &zip :: BTree\ \alpha\ \beta \to BTree\ \gamma\ \delta \to BTree\ (\alpha, \gamma)\ (\beta, \delta) \\ &zip\ xt\ xt' = \mathbf{let}\ ytt = partition\ xt \\ &\qquad\qquad\qquad\quad ytt' = partition\ xt' \\ &\qquad\qquad\quad \mathbf{in}\ \ flatten\ (map\ (local_z)\ (zip\ ytt\ ytt')) \end{aligned}$$

where $local_z$ has the type of

$$\begin{aligned} local_z :: &(\alpha \to \alpha \to BTree\ \alpha\ \beta) \to (\gamma \to \gamma \to BTree\ \gamma\ \delta) \\ &\to (\alpha, \gamma) \to (\alpha, \gamma) \to BTree\ (\alpha, \gamma)\ (\beta, \delta). \end{aligned}$$

$local_z$ recieves two subtrees with holes and zip them up. If two subtrees have no holes, $local_z$ is the same as *zip*.

distributed form of reduce

The computation of reduce for the partitioned subtrees is done by first locally composing two-argument functions, and substituting the values of child subtrees for the arguments of those functions globally in a bottom-up manner. Distributive form of reduce is as follows.

$$\begin{aligned} &\mathit{reduce} :: (\alpha \to \gamma, \beta \to \gamma \to \gamma \to \gamma) \to \mathit{BTree}\ \alpha\ \beta \to \gamma \\ &\mathit{reduce}\ f = \mathit{reduce}\ \mathit{subst}_u \circ \mathit{map}\ (\mathit{local}_r\ f) \circ \mathit{partition} \end{aligned}$$

where

$$\begin{aligned} &\mathit{subst}_u && = (\mathit{subst}_u L, \mathit{subst}_u N) \\ &\mathit{subst}_u L\ (\mathit{Leaf}\ n) && = n\ _\ _ \\ &\mathit{subst}_u N\ (\mathit{Node}\ n\ l\ r) && = n\ l\ r \end{aligned}$$

and $\mathit{local}_r\ f$ has the type of

$$\mathit{local}_r\ f :: (\alpha \to \alpha \to \mathit{BTree}\ \alpha\ \beta) \to \gamma \to \gamma \to \gamma.$$

local_r applies given function to each node in a bottom-up manner just like reduce, but this computation is done by composing two-argument functions. To derive these function compositions from original function f is a difficult problem, and it will be explained in Section 5.1.

distributive form of upwards accumulate

Similar to reduce, the computation of upwards accumulate on partitioned subtrees is done by composing the two-argument functions locally, and substituting the values of child subtrees for the arguments of those functions globally, then obtaining the results locally. Ditributed form of upwards accumulate is as follows.

$$\begin{aligned} &\mathit{reduce} :: (\alpha \to \gamma, \beta \to \gamma \to \gamma \to \gamma) \to \mathit{BTree}\ \alpha\ \beta \to \mathit{BTree}\ \gamma\ \gamma \\ &\mathit{uAcc}\ f\ \mathit{xt} = \mathbf{let}\ \mathit{ytt} = \mathit{map}\ (\mathit{local}_u\ f)\ (\mathit{partition}\ \mathit{xt}) \\ &\qquad\qquad\ \ \mathit{zt} = \mathit{getchildren}\ (\mathit{uAcc}\ \mathit{subst}_u\ (\mathit{map}\ \mathit{root}\ \mathit{ytt})) \\ &\qquad\quad \mathbf{in}\ \ \mathit{flatten}\ (\mathit{map}\ (\lambda(t,(l,r)).\mathit{map}\ (\lambda n.n\ l\ r))\ (\mathit{zip}\ \mathit{ytt}\ \mathit{zt})) \end{aligned}$$

where

$$\begin{aligned} &\mathit{subst}_u && = (\mathit{subst}_u L, \mathit{subst}_u N) \\ &\mathit{subst}_u L\ (\mathit{Leaf}\ n) && = n\ _\ _ \\ &\mathit{subst}_u N\ (\mathit{Node}\ n\ l\ r) && = n\ l\ r \\ &\mathit{getchildren}\ (\mathit{Leaf}\ n) && = (\bot, \bot) \\ &\mathit{getchildren}\ (\mathit{Node}\ n\ l\ r) && = (\mathit{rootl}, \mathit{rootr}) \end{aligned}$$

and $\mathit{local}_u\ f$ has the type of

$$\mathit{local}_u\ f :: (\alpha \to \alpha \to \mathit{BTree}\ \alpha\ \beta) \to F\ \gamma \to F\ \gamma \to \mathit{BTree}\ (F\ \gamma)\ (F\ \gamma)$$

where

$$\text{data}\ F\ \gamma = \gamma \to \gamma \to \gamma.$$

local_u applies given function to each node in a bottom-up manner just like upwards accumulate, but this computation is done by composing two-argument functions. To derive these function compositions is also a difficult problem, and it will be explained in Section 5.1.

distributive form of downwards accumulate

The computation of downwards accumulate on partitioned subtrees is done by composing the one-argument functions locally, and substituting the value of the parent subtree for the argument of those functions globally in a top-down manner, then obtaining the result locally. Ditributive form of downwards accumulate is as follows.

$$
\begin{aligned}
&dAcc :: (\beta \to \gamma \to \gamma, \beta \to \gamma \to \gamma) \to BTree\ \alpha\ \beta \to BTree\ \gamma\ \gamma \\
&Acc\ f\ xt\ c = \mathbf{let}\ ytt = map\ local_d\ (partition\ xt) \\
&\qquad\qquad\qquad\quad zt = dAcc\ subst_d\ (map\ \lambda n.(left\ n, right\ n)\ ytt)\ c \\
&\qquad\qquad\quad \mathbf{in}\ \ flatten\ (map\ (\lambda(t, c').map\ (\lambda n.n\ c'))\ (zip\ ytt\ zt)).
\end{aligned}
$$

where

$$
\begin{aligned}
subst_d\ (l, r) &= (subst_{dL}\ l, subst_{dR}\ r) \\
subst_{dL}\ l\ c\ &= l\ c \\
subst_{dR}\ r\ c\ &= r\ c
\end{aligned}
$$

and $local_d\ f$ has the type of

$$
local_d\ f :: (\alpha \to \alpha \to BTree\ \alpha\ \beta) \to G\ \gamma \to G\ \gamma \to BTree\ \ (G\ \gamma)
$$

where

$$
\text{data}\ G\ \gamma = \gamma \to \gamma.
$$

$local_d$ computes propagating accumulate parameters in a top-down manner just like downwards accumulate, but this computation is done by composing one-argument functions. To derive these function compositions is also a difficult problem, and it will be explained in Section 5.2.

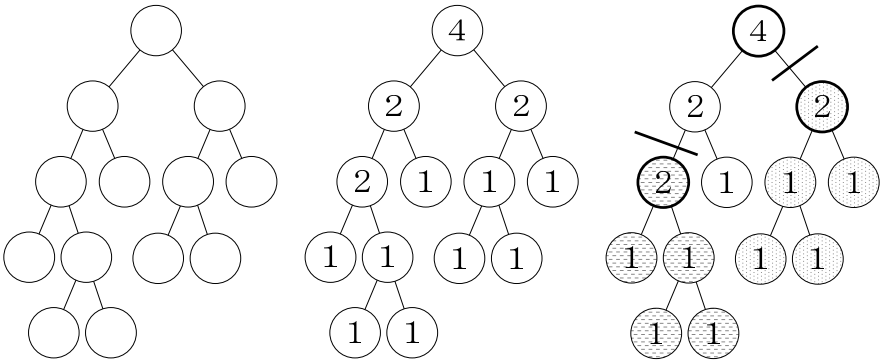
4 Implementation of Tree Skeletons

4.1 Tree partition

The naive strategy to partition the tree is to iterate to chop at the root of subtree, until the expected number of components. However, if the original tree is imbalance, the sizes of subtrees obtained by this strategy differ so much that the computation will be inefficient.

We use the m-bridge technique[7] to partition trees. Fig. 2 shows an example of partition of a tree by m-bridge technique. The advantages of this method are as follows:

- *simple algorithm* To partition with m-bridge technique, we only compute the $weight/m$ for each node where *weight* is the size of subtree whose root is the node, and m is a given value, and compare them.
- *connection* All nodes distributed to each processor are connected, and this feature enables us to decrease the communication between processors.



(a) original tree (b) compute *weight*/m (c) partitioned tree

Fig. 2. partition by m-bridge technique ($m = 3$)

- *almost same size* The number of nodes of each subtree is not more than $4n/p$ where n is the number of nodes of the original tree and p is the number of processors. This feature assures the load-balancing.
- *same shape* If the original tree is a binary tree, the relation of destributed subtrees is also a binary tree. Furthermore, the number of nodes where the child subtrees have connected is at most one in the left descendants of root, and at most one in the right descendants. This feature is very important to consider the applicability of our skeletons.

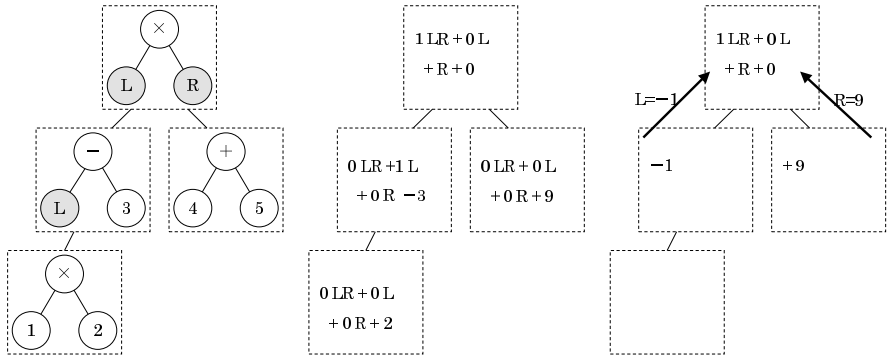
4.2 Implementation of reduce

In this subsection, we describe the details of our implementation with an example of reduce on a tree. We consider to evaluate a calculation tree whose leaves are numbers, and whose internal nodes are functions of $(+)$, $(-)$ and $(\times)$. With reduce, a parallel skeleton program to evaluate this calculation tree is as follows,

$$
\begin{aligned}
&\mathit{evaltree} :: (\mathit{BTree}\ \mathit{Num}\ (\mathit{Num} \to \mathit{Num} \to \mathit{Num}) \to \mathit{Num}\\
&\mathit{eval} :: (\mathit{Num} \to \mathit{Num} \to \mathit{Num}) \to \mathit{Num} \to \mathit{Num} \to \mathit{Num}\\
&\mathit{evaltree} = \mathit{reduce}\ (\mathit{id}, \mathit{eval})\\
&\qquad\qquad \textbf{where}\ \mathit{eval}\ n\ l\ r = n\ l\ r
\end{aligned}
$$

After partitioning an original tree (Fig. 3 (a)), reduction on the partitioned tree is computed by two steps, (1) local function composition and (2) global substitution.

Local function composition In this step, we compute inside of each subtree independently by composing functions in a bottom-up manner. In my implementaion with C++ language, these functions are represented by instances of a



(a) after partition (b) local reduction (c) global reduction

Fig. 3. implementation of reduce

structures and their compositions are represented by calcurations on those instances. Currently, those definitions of the structure and calculations are given by user.

In the example of calculation tree, the functions of all nodes can be represent as $\lambda LR.aLR + bL + cR + d$ where L, R are the holes of the subtrees. Fig. 3 (b) shows the result of local function compositions.

Global substitution In this step, we compute globally by substituting the results for the arguments of the functions by communicating between processors. We substite the results of two child subtrees into L and R of parent subtree's function in a bottom-up manner.

In Fig. 3 (c), substitution of two values -1 and 9 for the arguments of root function is being carried out.

4.3 Experiments

We implemented tree skeletons with C++ language and MPI library, and made experiments as follows.

- We made a program with map, reduce and upwadsaccumulate skeletons.
- The number of nodes of each tree was 1,000,000 and they were randomly generated to be imbalanced.
- We used PC-clusters with changing the number of processors from 1 to 12.

The results for five trees are shown in Fig. 4. Note that the results are shown in relative performance except partitioning and flattening, and the figure contains two bound lines, one is ideal maximum of speedup (the line when the cost is n/p), and the other is the quarter of ideal maximum of speedup (the line when the cost is $4n/p$, the upper bound of number of nodes by m-bridge partition). It can be seen from the results that our implementation achieves 5 times speedup by using 12 processors, and the speedup is no less than the lower bound.

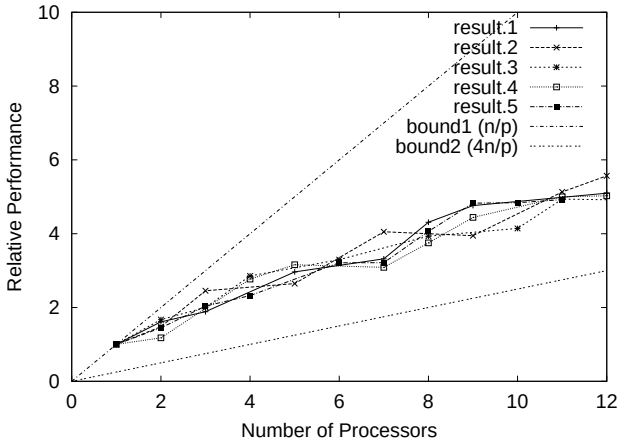


Fig. 4. relative speedup with 1–12 processors

5 Conditions for Efficient Implementations

As we saw in Section 3, it is necessary to derive local function compositions, but this is not self-apparent. In this section, we will present the conditions to efficiently implement tree skeletons, according to the idea of*context preservation*[2].

5.1 reduce and upwards accumulate

In the case of reduce and upwards accumulate, each subtree has two holes, so the functions that are the results of local computation have two arguments, but their compositions are dificcult.

However, from the feature of partition by the m-bridge, each node except the root has only one descendant, which is a hole, and the functions of local computation have only one argument in this case. This enables us to obtain the conditions for efficient implementation as follows.

Theorem 2 (conditions for efficient implementation of reduce and upwards accumulate). Two skeletons *reduce* (f_L, f_N) and *uAcc* (f_L, f_N) can be efficiently implemented if the function f_N satisfies the following conditions.

Let

$$f_N = \lambda l.\lambda r.C[l, r, e]$$

where $C[l, r, e]$ be a finite-length context with two arguments, and e includes no l and r. By assuming l or r as a constant value, we obtain two contexts with one arguments,

$$C[l, r, e] \to \lambda C_1[r, e'], \lambda C_2[l, e''].$$

We can obtain an efficient implementation of reduce or upwards accumulate if these two context $C_1[x, e]$ and $C_2[x, e]$ are closed in the sense that for arbitrary $a \in \{1, 2\}, b \in \{1, 2\}$, the following formula is satisfied,

$$\lambda x.C_a[x, e] \circ \lambda x'.C_b[x', e'] = \lambda x'.C_b[x', E(e, e')]$$

where $E(e, e')$ stands for a expression of e and e'. □

To illustrate this condition, we explain with an example function that is an extention of *reduce* $(id, sum3)$ described in Section 2.

Example 1. Let a function f_N be defined as

$$f_N\ n\ l\ r = l \oplus n \oplus r$$

where $\oplus$ is an associative and commutative operator. By abstracting function f_N, we obtain the following context:

$$C[l, r, e] = l \oplus e \oplus r.$$

Assuming one of arguments as a constant value, we can obtain two contexts with one argument,

$$C_1[x.e] = e \oplus x \quad \text{and} \quad C_2[x, e] = x \oplus e.$$

These two contexts are closed because for example one of compositions is as follows,

$$\begin{aligned} \lambda x.C_1[x, e] \circ \lambda x'.C_2[x', e'] &= \lambda x.(e \oplus x) \circ \lambda x'.(x' \oplus e') \\ &= \lambda x'.e \oplus (x' \oplus e') \\ &= \lambda x'.x' \oplus (e \oplus e') \\ &= \lambda x'.C_2[x', e \oplus e'], \end{aligned}$$

and the other compositions also can be proved with similar way. Therefore, reduce and upwards accumulate with f_N can be efficiently implemented with our approach.

5.2 downwards accumulate

In the case of downwards accumulate, each subtree has an unknown information, which is the result of its parent subtree. Therefore, we must compute composition of functions that have one argument. With the idea of context preservation, we obtain the following conditions for efficient implementation of downwards accumulate.

Theorem 3 (conditions for efficient implementation of downwards accumulate). The skeleton $dAcc\ (f_L, f_R)$ can be efficiently implemented if the functions f_L and f_R satisfy the following conditions.

Let

$$f_L = \lambda x.C_L[x, e] \quad \text{and} \quad f_R = \lambda x.C_R[x, e]$$

where $C_L[x, e]$ and $C_R[x, e]$ be finite-length contexts with one argument, and e includes no x. We can obtain an efficient implementation of downwards accumulate if there is a context $C[x, e]$, which satisfies

$$C[x, e] \supseteq C_L[x, e] \quad \text{and} \quad C[x, e] \supseteq C_R[x, e],$$

and the context $C[x, e]$ is closed in the sense that the following formula is satisfied,

$$\lambda x.C[x, e] \circ \lambda x'.C[x', e'] = \lambda x'.C[x', E(e, e')]$$

where $E(e, e')$ stands for a expression of e and e'.

□

To illustrate this condition, we explain with an example function that is an extention of *dAcc* (h_L, h_R) described in the example of in-order numbering problem in Section 2.

Example 2. Let functions h_L and h_R be defined as

$$h_L\ n\ c = c \quad \text{and} \quad h_R\ n\ c = n \oplus c$$

where $\oplus$ is an associative operator. By abstracting two functions, we obtain the following contexts:

$$C_L[x, e] = x \quad \text{and} \quad C_R[x, e] = x \oplus e.$$

The context $C[x, e] = x \oplus e$ is a superior set of $C_L[x, e]$ and $C_R[x, e]$ and the context $C[x, e]$ is closed, because the composition is as follows,

$$\begin{aligned} \lambda x.C[x, e] \circ \lambda x'.C[x', e'] &= \lambda x.(x \oplus e) \circ \lambda x'.(x' \oplus e') \\ &= \lambda x'.(x' \oplus e') \oplus e \\ &= \lambda x'.x' \oplus (e' \oplus e) \\ &= \lambda x'.C[x', e' \oplus e], \end{aligned}$$

Therefore, we can obtain an efficient implementation of downwards accumulate with f_L and f_R.

6 Conclusion

In this paper, we have proposed a new implementation of tree skeletons, which are efficient on distributed systems. Our approach is as follows, first we partition a tree into subtrees with almost equal size and good shape by m-bridge technique, and locally compute by composing functions independently, then propagate the results over the tree. The parallel computation costs of our implementations on distributed systems are $O(n/p + \log p)$ and this cost is the same as tree contraction on shared memory systems. The results of some experiments shows that our implementation is efficient even for the imbalanced trees. Furthermore, we present the conditions for efficient implementation of our approach.

Our future work are as follows. The conditions presented in this paper may be too strict, and if we can loose the conditions, our approach will be a more powerful and general one. Furthermore, there are no environments for tree skeletons, we want to build an extended skeleton programming environment with our tree skeletons.

Acknowledgement

The authors thank to ToPS members for their useful comments for formalizing our ideas.

References

1. R.S. Bird: An introduction to the theory of list. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pp. 5–42, Springer-Verlag, 1987.
2. W. Chin, A. Takano, Z. Hu: Parallelization via Context Preservation. In *IEEE Computer Society International Conference on Computer Languages (ICCL'98)*, Loyola University Chicago, Chicago, USA, May 1998. IEEE Press.
3. M. Cole: *Algorithmic skeletons : a structured approach to the management of parallel computation*. Research Monographs in Parallel and Distributed Computing, 1989.
4. A. L. Fisher, A. M. Ghuloum: Parallelizing complex scans and reductions. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI'94)*, 1994.
5. J. Gibbons, W. Cai and D. B. Skillicorn: Efficient Parallel Algorithms for Tree Accumulations. *Science of Computer Programming*, vol 23, no. 1, pp. 1–18, 1994.
6. Z. Hu, M. Hakeich, H. Iwasaki: Diffusion: Calculating Efficient Parallel Programs. *1999 ACM SIGPLAN Workshop on Partial Evaluation and Symantics-Based Program Manipulation (PEPM'99)*, pp. 85–94, San Antonio, Texas, 1999.
7. M. Reid-Miller, G. L. Miller, F. Modugno: List Ranking and Parallel Tree Contraction. In *Synthesis of Parallel Algorithms*, chapter 3, pp. 115–194, Morgan Kaufmann Publishers, 1996.
8. D. B. Skillicorn: The Bird-Meertens Formalism as a Parallel Model. In *NATOARW Software for Parallel Computation*, June 1992.
9. D. B. Skillicorn: Parallel Implementation of Tree Skeletons. *Journal of Parallel and Distributed Computing*, vol 39, no. 2, pp. 115–125, 1996.
10. T. Takahashi, H. Iwasaki and Z. Hu: Efficient Parallel Skeletons for Nested Data Structures. In *Proceedings of The 2001 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2001)*, Monte Carlo Resort, Las Vegas, June 2001. CSREA Press.