MATHEMATICAL ENGINEERING
TECHNICAL REPORTS

Efficient Implementation of Tree Skeletons on Distributed-Memory Parallel Computers

Kiminori MATSUZAKI and Zhenjiang HU

METR 2006–65 December 2006

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

WWW page: http://www.i.u-tokyo.ac.jp/edu/course/mi/index_e.shtml

Efficient Implementation of Tree Skeletons on Distributed-Memory Parallel Computers

Kiminori Matsuzaki and Zhenjiang Hu

Department of Mathematical Informatics,
Graduate School of Information Science and Technology,
University of Tokyo
{kmatsu,hu}@mist.i.u-tokyo.ac.jp

Abstract. Parallel tree skeletons are basic computational patterns that encourage us to develop parallel programs manipulating trees. In this paper, we develop an efficient implementation of parallel tree skeletons on distributed-memory parallel computers. In our implementation, we divide a binary tree based on the idea of m-bridges to obtain high locality, and represent local segments as serialized arrays to obtain high sequential performance. We furthermore develop a cost model of our implementation of parallel tree skeletons. We confirmed the efficacy of our implementation with several experiments.

1 Introduction

Parallel tree skeletons, first formalized by Skillicorn [34, 35], are basic computational patterns of parallel programs manipulating trees. By using parallel tree skeletons, users can develop parallel programs without bothering the low-level implementation and the details of parallel computers. There are several studies on the systematic methods of developing parallel programs by means of parallel tree skeletons [9, 19, 21, 36, 37].

For efficient parallel tree manipulations, tree contraction algorithms have been studied intensively [1, 8, 24, 25, 38]. Many tree contraction algorithms have been developed on many parallel computational models, for instance, EREW PRAM [1], Hypercubes [24], and BSP/CGM [8]. While the original tree contraction algorithm, proposed by Miller and Reif [25], is a parallel algorithm that reduces a tree into the root by independent removals of nodes, several parallel tree manipulations are developed based on the tree contraction algorithms [1, 10]. For tree skeletons, Gibbons et al. [13] developed an implementation algorithm of parallel tree skeletons based on tree contraction algorithms.

In this paper, we develop an efficient implementation of parallel tree skeletons for binary trees on distributed-memory parallel computers. Compared with the implementations so far that mainly target shared-memory parallel computers, our implementation has the following three features.

- *Less overheads of parallelism.* Locality is one of the most important properties in developing efficient parallel programs especially for distributed-memory computers. We adopt m-bridges [31] in the basic graph-theory to divide binary trees with high locality. Furthermore, to minimize the overheads of parallelism, we formalized the tree skeletons as sequential functions with some auxiliary functions for parallel implementation.
- *High sequential performance.* The performance of the sequential parts is as important as that of the communication parts for efficient parallel programs. We represent a local segment as a serialized array and implemented local computations in the tree skeletons with loops rather than recursive functions. High sequential performance is obtained with these techniques.

– *Cost model.* We also formalize a cost model of our parallel implementation. The cost model helps us to divide binary trees with good load balance.

We have implemented parallel tree skeletons in C++ and MPI, and the skeletons are available as a part of the skeleton library SkeTo [22]. We confirmed the efficacy of our implementation of tree skeletons with several experiments.

This paper is organized as follows. In the following Section 2, we introduce parallel tree skeletons with two examples. In Section 3, we discuss the division of binary trees after reviewing basic graph-theoretic results. In Section 4, we develop an efficient implementation and a cost model of parallel tree skeletons on distributed-memory parallel computers. Based on this cost model, we discuss the optimal division of binary trees in Section 5. We then show several experiment results in Section 6. We review related work in Section 7, and finally we make concluding remarks in Section 8.

2 Parallel Tree Skeletons

2.1 Notations

In this paper, we borrow the notation of Haskell [4,30]. In the following, we briefly introduce important notations and the data structure of binary trees. Roughly speaking, the definitions in this paper can be read as mathematical function definitions except for the function applications denoted by spaces.

Functions and Operators Function application is denoted by a space and the argument may be written without brackets. Thus $f\ a$ means $f(a)$. Functions are curried, and the function application associates to the left. Thus $f\ a\ b$ means $(f\ a)\ b$. The function application binds stronger than any other operator, so $f\ a \oplus b$ means $(f\ a) \oplus b$, but does not $f\ (a \oplus b)$. The identity function is denoted by id.

Some arguments do not affect to the result of the functions. In such cases the arguments may be called *don't-care values* and they are donated as $\_$.

In addition to arithmetic operators we use binary operator $\uparrow$ that returns the larger of the two arguments. Operators can be sectioned and be treated as functions, that is, $a \oplus b = (\oplus)\ a\ b$ holds.

Binary Trees Binary trees are trees whose internal nodes have exactly two children. In this paper, leaves and internal nodes of a binary tree may have different types. The datatype of binary trees whose internal nodes have values of type α and leaves have values of type β is defined as follows.

$$\begin{aligned}\text{data } \mathsf{BTree}\ \alpha\ \beta = &\ \mathsf{BLeaf}\ \alpha \\ \mid &\ \mathsf{BNode}\ (\mathsf{BTree}\ \alpha\ \beta)\ \beta\ (\mathsf{BTree}\ \alpha\ \beta)\end{aligned}$$

We introduce function *root* that returns the value of the root node.

$$\begin{aligned}root\ (\mathsf{BLeaf}\ a) &= a \\ root\ (\mathsf{BNode}\ l\ b\ r) &= b\end{aligned}$$

2.2 Parallel Tree Skeletons

Parallel binary-tree skeletons (parallel tree skeletons in short) are basic computational patterns manipulating binary trees in parallel. In this section, we introduce a set of basic parallel tree skeletons first proposed by Skillicorn [34, 35] with minor modifications.

A set of basic parallel tree skeletons includes five higher-order functions categorized into the following three.

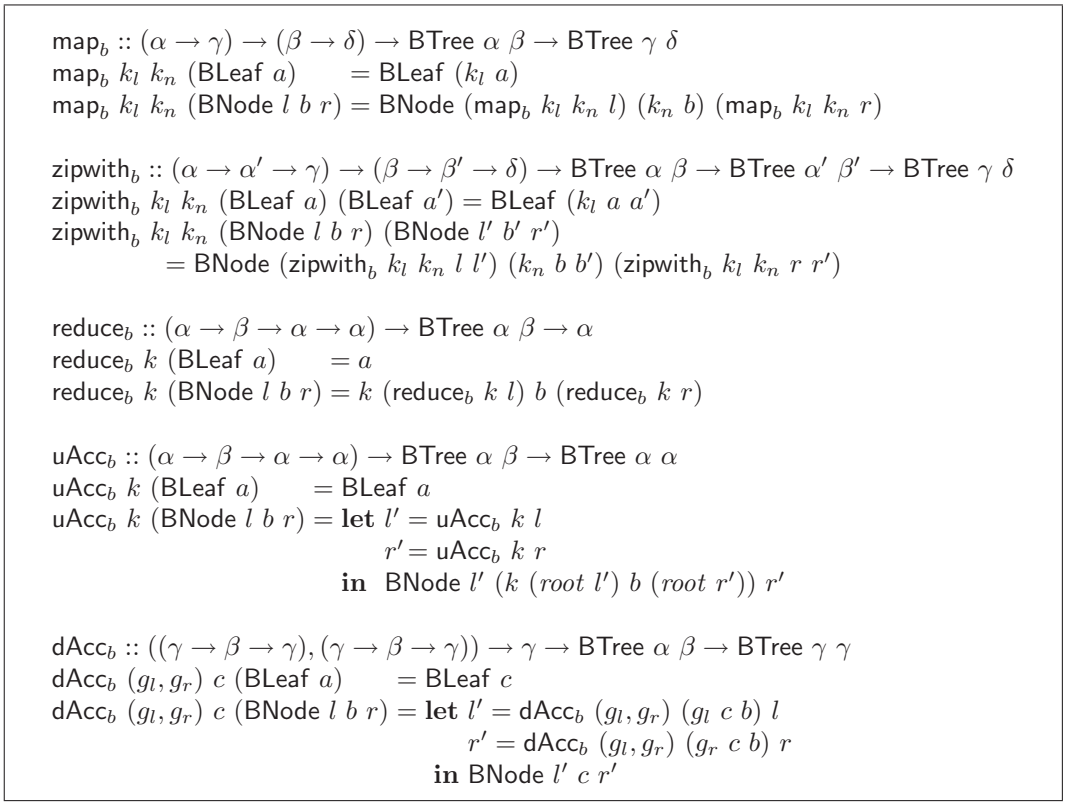


Fig. 1. Definition of parallel tree skeletons.

– Node-wise computations: *map* and *zipwith*
  The parallel skeleton map_b takes two functions k_l and k_n and a binary tree, and applies k_l to each leaf and k_n to each internal node. The parallel skeleton $\mathsf{zipwith}_b$ takes two functions k_l and k_n and two binary trees of the same shape, and zips the trees up by applying k_l to each pair of leaves and k_n to each pair of internal nodes.

– Bottom-up computations: *reduce* and *upwards accumulate*
  The parallel skeleton reduce_b takes a function k and a binary tree, and collapses the tree into a value by applying the function k in a bottom-up manner. The parallel skeleton uAcc_b (upwards accumulate) also takes a function k and a binary tree, and computes $(\mathsf{reduce}_b\ k)$ for each subtree. In other words, the uAcc_b skeleton is a shape-preserving manipulation of trees where the resulting values are the intermediate results of the bottom-up reduction.

– Top-down computation: *downwards accumulate*
  The parallel skeleton dAcc_b (downwards accumulate) is another shape-preserving manipulation of trees. This skeleton takes two functions g_l and g_r, an accumulative parameter c and a binary tree, and computes a value for each node by updating the accumulative parameter c in a top-down manner. The update is done by function g_l for the left child, and by function g_r for the right child.

We give the formal sequential definition of these parallel tree skeletons in Fig. 1. We denote the parallel tree skeletons in the sans-serif font with a suffix b. Note that the definitions of the uAcc_b skeleton and the dAcc_b skeleton is different from those defined by Skillicorn [34, 35] in the sense that we defined them as recursive functions not in the point-free style programming.

To guarantee existence of efficient parallel implementations for many parallel computers, the parallel tree skeletons require some conditions for their parameter functions. The map_b and $\mathsf{zipwith}_b$ skeletons require no condition. For the reduce_b, uAcc_b and dAcc_b skeletons, we formalize the conditions for parallel implementation as follows as existence of auxiliary functions satisfying a certain closure property.

The reduce_b and uAcc_b skeletons with parameter function k require existence of four auxiliary functions ϕ, ψ_n, ψ_l, and ψ_r satisfying the following equations.

$$
\begin{aligned}
&k\ l\ b\ r = \psi_n\ l\ (\phi\ b)\ r \\
&\psi_n\ (\psi_n\ x\ l\ y)\ b\ r = \psi_n\ x\ (\psi_l\ l\ b\ r)\ y \\
&\psi_n\ l\ b\ (\psi_n\ x\ r\ y) = \psi_n\ x\ (\psi_r\ l\ b\ r)\ y
\end{aligned}
$$

Intuitive meaning of these auxiliary functions is:

> For parallel computation we require some domain where there is a certain associative computation. The computation on an internal node is lifted up by function ϕ to the domain and pulled down by function ψ_n from the domain. The certain kind of associativity on the domain is given by functions ψ_l and ψ_r that satisfy the closure property.

We denote the function k satisfying the condition as $k = \langle \phi, \psi_n, \psi_l, \psi_r \rangle_u$.

The dAcc_b skeleton with parameter functions g_l and g_r requires existence of auxiliary functions ϕ_l, ϕ_r, ψ_u, and ψ_d satisfying the following equations.

$$
\begin{aligned}
&g_l\ c\ b = \psi_d\ c\ (\phi_l\ b) \\
&g_r\ c\ b = \psi_d\ c\ (\phi_r\ b) \\
&\psi_d\ (\psi_d\ c\ b)\ b' = \psi_d\ c\ (\psi_u\ b\ b')
\end{aligned}
$$

Intuitive meaning of these auxiliary functions is:

> For parallel computation we require some domain in which there is an associative computation. The computation on an internal node is lifted up by functions ϕ_l and ϕ_r to the domain and pulled down by function ψ_d. The function ψ_u indicates the associative computation in the domain.

We denote the pair of functions (g_l, g_r) satisfying the condition as $(g_l, g_r) = \langle \phi_l, \phi_r, \psi_u, \psi_d \rangle_d$.

2.3 Examples

To illustrate how we can develop parallel programs by composing these parallel tree skeletons, we show skeletal parallel programs for two examples, computing height and the party planning problem [7].

Computing Height of Binary Tree Height of a binary tree is the maximum of depths for all the nodes. Since the depths can be computed by using dAcc_b skeletons, we can develop a skeletal parallel program that computes the height of a binary tree as follows. In this definition, the auxiliary functions are easily derived because the parameter functions for uAcc_b and dAcc_b skeletons are defined with an associative operator, respectively.

$$
\begin{aligned}
\mathit{height}\ t = \ &\mathbf{let}\ \mathit{dt} = \mathsf{dAcc}_b\ ((+),(+))\ 1\ t \\
&\mathbf{in}\ \mathsf{reduce}_b\ \mathit{max3}\ \mathit{dt} \\
&\mathbf{where}\ \mathit{max3}\ l\ b\ r = l \uparrow b \uparrow r \\
&\qquad ((+),(+)) = \langle \mathit{id}, \mathit{id}, (+), (+) \rangle_d \\
&\qquad \mathit{max3} = \langle \mathit{id}, \mathit{max3}, \mathit{max3}, \mathit{max3} \rangle_u
\end{aligned}
$$

In fact, we can develop another skeletal parallel program that computes the height of a binary tree with a single bottom-up computation. The following recursive function computes the height of a binary tree with a single bottom-up computation.

$$\begin{aligned} &\mathit{height}\ (\mathsf{BLeaf}\ a) &&= 1 \\ &\mathit{height}\ (\mathsf{BNode}\ l\ b\ r) &&= 1 + (l \uparrow r) \end{aligned}$$

By applying the parallelization techniques in [19] to this recursive function, we can obtain the following skeletal parallel program.

$$\begin{aligned} \mathit{height}\ t = {} & \mathsf{reduce}_b\ (\lambda l\ b\ r.1 + (l \uparrow r))\ (\mathsf{map}_b\ (\lambda x.1)\ \mathit{id}\ t) \\ & \textbf{where}\ \lambda l\ b\ r.1 + (l \uparrow r) = \langle \phi, \psi_n, \psi_l, \psi_r \rangle_u \\ & \qquad \phi\ b = (-\infty, b) \\ & \qquad \psi_n\ l\ (b_1, b_2)\ r = b_1 \uparrow (b_2 + l) \uparrow (b_2 + r) \\ & \qquad \psi_l\ (l_1, l_2)\ (b_1, b_2)\ r = (b_1 \uparrow (b_2 + l_1) \uparrow (b_2 + r), b_2 + l_2) \\ & \qquad \psi_r\ l\ (b_1, b_2)\ (r_1, r_2) = (b_1 \uparrow (b_2 + l) \uparrow (b_2 + r_1), b_2 + r_2) \end{aligned}$$

As seen in this program, we often require more computation in the auxiliary functions of the skeletons than sequential programs. The complicity of auxiliary functions can be considered as overheads for parallel computation.

Party Planning Problem The party planning problem appeared in a textbook [7] as an exercise for sequential dynamic programming problem on trees. The specification of the party planning problem is as follows.

> 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.

This problem is an instance of so-called maximum marking problems [5, 32].

A known sequential program that solves the party planning problem is given as the function *ppp* with auxiliary functions *ppp′* and *maxsums* as shown in Fig. 2. In the program, the function *maxsums* takes a binary tree and computes a pair of values (ms, us):

- ms: the maximum of sums of non adjacent nodes under the condition that the root node is selected, and
- us: the maximum of sums of non adjacent nodes under the condition that the root node is not selected.

From the sequential program, we can obtain a skeletal parallel program as shown in Fig. 3 by applying the derivation techniques in our previous papers [19, 21]. The detailed derivation of the skeletal parallel program will be shown in the first author's Ph.D. thesis.

$$
\begin{array}{l}
ppp :: \mathsf{BTree}\ Int \to \mathsf{BTree}\ \mathsf{Bool} \\
ppp\ t = ppp'\ False\ t \\
ppp'\ p\_marked\ (\mathsf{BLeaf}\ a) \quad = \textbf{if}\ p\_marked\ \textbf{then}\ \mathsf{BLeaf}\ False \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{else let}\ (ms, us) = maxsums\ (\mathsf{BLeaf}\ a) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ \mathsf{BLeaf}\ (ms > us) \\
ppp'\ p\_marked\ (\mathsf{BNode}\ l\ b\ r) = \textbf{if}\ p\_marked \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{then}\ \mathsf{BNode}\ (ppp'\ False\ l)\ False\ (ppp'\ False\ r) \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{else let}\ (ms, us) = maxsums\ (\mathsf{BNode}\ l\ b\ r) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad marked = ms > us \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ \mathsf{BNode}\ (ppp'\ marked\ l)\ marked\ (ppp'\ marked\ r) \\
maxsums\ (\mathsf{BLeaf}\ a) \quad = (a, 0) \\
maxsums\ (\mathsf{BNode}\ l\ b\ r) = \textbf{let}\ (ms_l, us_l) = maxsums\ l \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (ms_r, us_r) = maxsums\ r \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ (us_l + b + us_r, (ms_l \uparrow us_l) + (ms_r \uparrow us_r))
\end{array}
$$

Fig. 2. A sequential program that solves the party planning problem.

$$
\begin{array}{l}
ppp\ t = \textbf{let}\ t' = \mathsf{uAcc}_b\ \langle \phi^u, \psi^u_n, \psi^u_l, \psi^u_r \rangle_u\ (\mathsf{map}_b\ (\lambda a.(a, 0))\ id\ t) \\
\qquad\qquad ct = \mathsf{dAcc}_b\ \langle \phi^d, \phi^d, \psi^d_u, \psi^d_d \rangle_d\ False\ t' \\
\qquad\quad \textbf{in}\ \mathsf{zipwith}_b\ mark\ mark\ ct\ t' \\
\\
mark\ c\ (ms, us) = \textbf{if}\ c\ \textbf{then}\ False\ \textbf{else}\ ms > us \\
\\
\phi^u\ b = \left(\begin{pmatrix} 0 & -\infty \\ -\infty & 0 \end{pmatrix}, b \right) \\
\\
\psi^u_n\ \begin{pmatrix} l_1 \\ l_2 \end{pmatrix}\ \left(b, \begin{pmatrix} a_{11}\ a_{12} \\ a_{21}\ a_{22} \end{pmatrix} \right)\ \begin{pmatrix} r_1 \\ r_2 \end{pmatrix} \\
\qquad = \textbf{let}\ \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b + l_2 + r_2 \\ (l_1 \uparrow l_2) + (r_1 \uparrow r_2) \end{pmatrix}\ \textbf{in}\ \begin{pmatrix} (a_{11} + x_1) \uparrow (a_{12} + x_2) \\ (a_{21} + x_1) \uparrow (a_{22} + x_2) \end{pmatrix} \\
\\
\psi^u_l\ \left(b^l, \begin{pmatrix} a^l_{11}\ a^l_{12} \\ a^l_{21}\ a^l_{22} \end{pmatrix} \right)\ \left(b_n, \begin{pmatrix} a^n_{11}\ a^n_{12} \\ a^n_{21}\ a^n_{22} \end{pmatrix} \right)\ \begin{pmatrix} r_1 \\ r_2 \end{pmatrix} \\
\qquad = \left(b^l, \begin{pmatrix} a^n_{11}\ a^n_{12} \\ a^n_{21}\ a^n_{22} \end{pmatrix} \times_{+,\uparrow} \begin{pmatrix} -\infty & b^n + r_2 \\ r_1 \uparrow r_2 & r_1 \uparrow r_2 \end{pmatrix} \times_{+,\uparrow} \begin{pmatrix} a^l_{11}\ a^l_{12} \\ a^l_{21}\ a^l_{22} \end{pmatrix} \right) \\
\\
\psi^u_r\ \begin{pmatrix} l_1 \\ l_2 \end{pmatrix}\ \left(b_n, \begin{pmatrix} a^n_{11}\ a^n_{12} \\ a^n_{21}\ a^n_{22} \end{pmatrix} \right)\ \left(b^r, \begin{pmatrix} a^r_{11}\ a^r_{12} \\ a^r_{21}\ a^r_{22} \end{pmatrix} \right) \\
\qquad = \left(b^r, \begin{pmatrix} a^n_{11}\ a^n_{12} \\ a^n_{21}\ a^n_{22} \end{pmatrix} \times_{+,\uparrow} \begin{pmatrix} -\infty & b^n + l_2 \\ l_1 \uparrow l_2 & l_1 \uparrow l_2 \end{pmatrix} \times_{+,\uparrow} \begin{pmatrix} a^r_{11}\ a^r_{12} \\ a^r_{21}\ a^r_{22} \end{pmatrix} \right) \\
\\
\phi^d\ (ms, us) = (False, (ms > us)) \\
\\
\psi^d_d\ c\ (b_1, b_2) = \textbf{case}\ c\ \textbf{of}\ \ True \to b_1; False \to b_2;
\end{array}
$$

(b_1, b_2) \ $\psi^d_u\ (b_1, b_2)\ (b'_1, b'_2)$	$(True, True)$	$(True, False)$	$(False, True)$	$(False, False)$
$(True, True)$	$(True, True)$	$(True, True)$	$(False, False)$	$(False, False)$
$(True, False)$	$(True, True)$	$(True, False)$	$(False, True)$	$(False, False)$
$(False, True)$	$(True, True)$	$(False, True)$	$(True, False)$	$(False, False)$
$(False, False)$	$(True, True)$	$(False, False)$	$(True, True)$	$(False, False)$

Fig. 3. The skeletal parallel program for the party planning problem. The function ψ^d_u looks up the table. The operator $\times_{+,\uparrow}$ is matrix multiplication on the commutative semi-ring $\{\mathsf{Num}, \uparrow, +\}$ where operators $+$ and $\uparrow$ are used instead of $\times$ and $+$ in the usual matrix multiplication.

3 Division of Binary Trees with High Locality

To develop efficient parallel programs on distributed-memory parallel computers, we need to divide data structures into smaller parts to distribute them to the processors. Here, the division of data structures should have the following two properties for efficiency of the parallel programs.

- *Locality.* The data distributed to each processor should be adjacent. If two elements that are adjacent in the original data are distributed to different processors, we may need communications between the processors.
- *Load balance.* The number of nodes distributed to each processor should be equal since the cost of local computation is often proportional to the number of nodes.

It is easy to divide a list with these two properties, that is, for a given list of N elements we simply divide the list into P sublists with N/P elements for each sublist. It is, however, difficult to divide a tree satisfying both of the two properties. The non linear and ill-balanced structure of binary trees makes it difficult to divide the tree into connected components with good load balance.

In this section, we introduce a division of binary trees based on the basic graph theory, and then we show the representation of distributed tree structure.

3.1 Graph-Theoretic Results for Division of Binary Trees

We start the discussion by introducing some graph-theoretic results [31]. Let $size_b(v)$ denote the number of nodes in the subtree rooted at node v.

Definition 1 (m-Critical Node [31]). Let m be an integer such that $1 < m \leq N$ where N is the number of nodes in a binary tree. A node v is called m-critical node, if v is an internal node and for each child v' of v inequality $\lceil size_b(v)/m \rceil > \lceil size_b(v')/m \rceil$ holds. □

Definition 2 (m-Bridge [31]). Let m be an integer such that $1 < m \leq N$ where N is the number of nodes in a binary tree. A set of maximal adjacent nodes where m-critical nodes are only on the terminals of the set is called an m-bridge. □

Figure 4 illustrates the m-critical nodes and the m-bridges.

The m-critical nodes and the m-brides have several properties that are important in dividing binary trees.

The following two lemmas show properties of the m-critical nodes and the m-bridges in terms of the global shape of them.

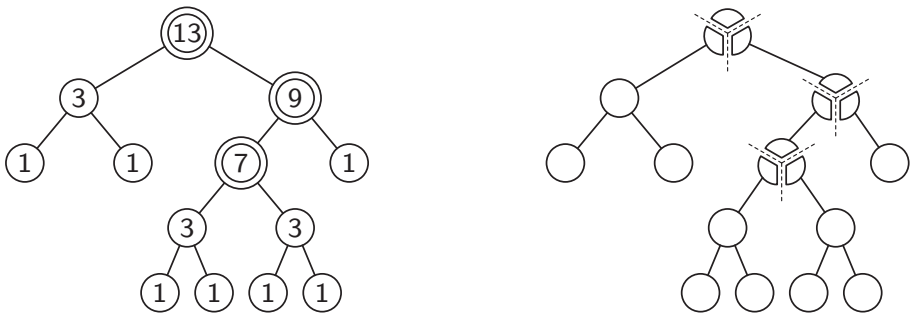


Fig. 4. An example of m-critical nodes and m-bridges. Left: In this binary tree, there are three 4-critical nodes denoted by the doubly-lined circles. The number in each node denotes the number of nodes in the subtree. Right: For the same tree there are seven 4-bridges, each of which is a set of connected nodes.

Lemma 1 ([31]). *If v_1 and v_2 are m-critical nodes then their least common ancestor is also an m-critical node.* □

Lemma 2 ([31]). *If B is an m-bridge of a tree then B has at most one m-critical node at the bottom.* □

The root node in each m-bridge is an m-critical node except for the root m-bridge that includes the global root node. If we remove the root m-critical node if it exists, from Lemma 2 and the definition of the m-bridge, the m-bridge has at most one m-critical node. In the following, we call the m-critical node in a segment as the *terminal node*.

The following three lemmas are related to the number of nodes in an m-bridge and the number of m-bridges in a tree. Note that the former two lemmas holds on general trees while the last lemma only holds on binary trees.

Lemma 3 ([31]). *The number of nodes in an m-bridge is at most $m + 1$.* □

Lemma 4 ([31]). *Let N be the number of nodes in a tree then the number of m-critical nodes in the tree is at most $2N/m - 1$.* □

Lemma 5. *Let N be the number of nodes in a binary tree then the number of m-critical nodes in the binary tree is at least $(N/m - 1)/2$.*

Proof. Let n_k be the number of nodes in binary trees that have k m-critical nodes. We prove this lemma by showing that the following inequality.

$$n_k \leq (2k + 1)m \tag{1}$$

holds by induction.
1. Base case ($k = 0$):

By definition of m-critical nodes, for the root node v we have $\lceil size(v)/m \rceil = 1$. Therefore, we obtain $0 < size(v) \leq m$, which satisfies the inequality (1) for the case $k = 0$.
2. Inductive case:

Assume that for all i such that $i < k$ inequality $n_i \leq (2i+1)m$ holds. Let v be the critical node nearest to the root node. Since the least common ancestor of two m-critical nodes is also m-critical node as Lemma 1 says, we can find such an m-critical node for any binary tree. Now we consider the following three parts of a tree: the left subtree of the node v, which has k_1 terminal nodes, the right subtree of the node v, which has k_2 terminal nodes, and the other parts, which has no terminal node. By definition $1 + k_1 + k_2 = k$ holds.

Let x_1, x_2, and x_3 be the numbers of nodes of the first, second, and third parts, respectively. Then, by hypothesis we obtain $x_1 \leq (2k_1 + 1)m$ and $x_2 \leq (2k_2 + 1)m$ hold. The number of nodes in the third part is at most m, that is $x_3 \leq m$, where the equality holds if the numbers of nodes v and root r are given as $size(v) = am + 1$ and $size(r) = (a + 1)m$ for some value a.

With these inequalities, we can prove the inequality (1) with the following calculation.

$$\begin{aligned} n_k &= x_1 + x_2 + x_3 \\ &\leq (2k_1 + 1)m + (2k_2 + 1)m + m \\ &= (2(k_1 + k_2 + 1) + 1)m \\ &= (2k + 1)m \end{aligned}$$

It follows from the transformation of inequality (1) as

$$\begin{aligned} n_k &\leq (2k + 1)m \\ (n_k/m - 1)/2 &\leq k \end{aligned}$$

that the lemma holds. □

In the previous studies [18,31], we divided a tree into m-bridges using the parameter m given by $m = 2N/P$ where N denotes the number of nodes and P denotes the number of processors. By this division we obtain at most $2P - 1$ m-bridges and thus each processor deals with at most two m-bridges in this case. This division of course enjoys high locality, but it is not good enough in terms of load balancing since the maximum number of nodes passed to a processor may be $2N/P$, which is twice of the average number of nodes N/P.

In Section 5, we adjust the value m for more efficient division based on the cost model developed in Section 4. The idea is that we divide a binary tree into more m-bridges using smaller m so that we obtain enough load balance while keeping the overheads caused by loss of locality rather small.

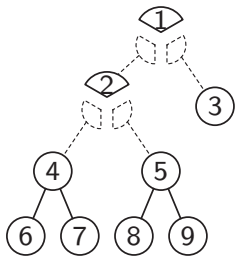
3.2 Data Structure for Distributed Segments

To obtain efficient parallel programs, the performance of the sequential parts is as important as that of the communication parts. This means that the data structure of local segments is important.

Generally speaking, data structure of trees are often implemented using pointers or references. There are, however, two problems in this implementation for large-scale tree applications. First problem is that a lot of memory is required. Considering trees of integers or trees of real numbers, for example, the pointers use as many memory as the value for each node. Furthermore, if we allocate nodes one by one, more memory are consumed for the information of freeing the nodes. Second problem is the loss of locality. Recent computers have a cache hierarchy to bridge the gap between the CPU speed and the memory speed, and cache misses greatly decrease the performance especially in data-intensive applications. If we allocate nodes from here and there then the probability of cache misses increase.

To resolve these problems, we represent a binary tree as an array serialized in the order of the preorder traversal. We represent a tree divided based on the m-bridges with one array gt for the global structure and one array of arrays $segs$ for the local segments. Note that the arrays in $segs$ are distributed among processors and only one processor has the array for each local segment. Figure 5 illustrates the array representation of the distributed tree. Since adjoining elements are aligned one next to another in this representation, we can reduce cache misses.

In the discussion of implementation algorithms in the next section, we denote $seg[i]$ for the ith value in the serialized array seg, and use functions isLeaf($seg[i]$), isNode($seg[i]$) and isTerminal($seg[i]$) to check whether the ith node is a leaf, an internal node, and a terminal node, respectively.



$$gt = [\square^{\mathsf{N}}, \square^{\mathsf{N}}, \square^{\mathsf{L}}, \square^{\mathsf{L}}, \square^{\mathsf{L}}]$$

$$segs = [\,[\,1^{\mathsf{T}}\,],\ [\,2^{\mathsf{T}}\,],\ [\,4^{\mathsf{N}}, 6^{\mathsf{L}}, 7^{\mathsf{L}}\,],\ [\,5^{\mathsf{N}}, 8^{\mathsf{L}}, 9^{\mathsf{L}}\,],\ [\,3^{\mathsf{L}}\,]\,]$$

Fig. 5. Array representation of divided binary trees. Each local segment of $segs$ is distributed to one of processors and is not shared. Labels L, N and T denote a leaf, a normal internal node, and a terminal node, respectively. Each m-critical node is included in the parent segment.

4 Implementation and Cost Model of Tree Skeletons

In this section, we show the implementation and the cost model of the tree skeletons on distributed-memory parallel computers. We implement the local computations in tree skeletons using loops and stacks on the serialized arrays to reduce the cache misses. This is the most significant technique with which the parallel programs achieve high performance in the sequential parts of the algorithm.

We introduce several parameters for discussion of the cost model (Table 1). The computational time of function f executed with p processors is denoted by $t_p(f)$. Parameter N denotes the number of nodes, and P denotes the number of processors. Parameter m is used for m-critical nodes and m-bridges, and M denotes the number of segments after the division. For the ith segment, in addition to the parameter of the number of nodes L_i, we introduce parameter D_i indicating the depth of the critical node. Parameter c_α denotes the communication time for a value of type α.

The cost model for tree accumulations can be uniformly given in the following form:

$$\max_p \sum_{pr(i)=p} (L_i \times t_l + D_i \times t_d) + M \times t_m$$

where $pr(i)$ denotes the processor assigned to ith segment, and t_l, t_d, and t_m are certain parameters. The cost model consists of the maximum cost of the local computation and the cost of the global computation. The cost of the local computation is the summation of costs for all the segments assigned to a processor, where $(L_i \times t_l)$ indicates the computational time required in sequential computation and $(D_i \times t_d)$ indicates the overheads for parallel computing. The last term $(M \times t_m)$ indicates the overheads of global computation.

4.1 Implementation and Cost Model of Map and Zipwith Skeleton

Since there are no dependencies among nodes in the computation of the map_b skeleton, we can implement the map_b skeleton by applying the following function MAP_LOCAL to each local segment. The MAP_LOCAL function applies function k_l to each leaf and function k_n to each internal node and the terminal node in a local segment seg.

```
MAP_LOCAL(k_l, k_n, seg)
    for i ← 0 to seg.size − 1
        if (isLeaf(seg[i]))        seg'[i] ← k_l(seg[i]);
        if (isNode(seg[i]))        seg'[i] ← k_n(seg[i]);
        if (isTerminal(seg[i]))    seg'[i] ← k_n(seg[i]);
    return seg';
```

In a local segment with L_i nodes, the number of leaves is at most $L_i/2+1$ and the number of internal nodes including the terminal node is at most $L_i/2+1$. Therefore, ignoring small

Table 1. Parameters for the cost model.

$t_p(f)$	computational time of function f using p processors
N	the number of nodes in the input tree
P	the number of processors
m	the parameter for m-critical nodes and m-bridges
M	the number of segments given by division of trees
L_i	the number of nodes in the ith segment
D_i	the depth of the terminal node in the ith segment
c_α	the time need for communicating one data of type α

constants we can specify the computational cost of the MAP_LOCAL function as follows.

$$t_1(\text{MAP\_LOCAL}) = \frac{L_i}{2} \times t_1(k_l) + \frac{L_i}{2} \times t_1(k_n)$$

Therefore, the cost model for the map_b skeleton is as follows.

$$t_P(\mathsf{map}_b) = \max_p \sum_{pr(i)=p} L_i \times \frac{t_1(k_l) + t_1(k_n)}{2}$$

Since the $\mathsf{zipwith}_b$ skeleton performs the similar computation as the map_b skeleton, we can give the implementation algorithm and the cost model for the $\mathsf{zipwith}_b$ skeleton in the same manner.

4.2 Implementation and Cost Model of Reduce Skeleton

We then show the implementation and the cost model of the reduce_b skeleton called with function k and auxiliary functions $k = \langle \phi, \psi_n, \psi_l, \psi_r \rangle_u$. Let the type of reduce_b skeleton be $\mathsf{reduce}_b :: (\beta \to \alpha \to \alpha \to \alpha) \to \mathsf{BTree}\ \alpha\ \beta \to \alpha$ and the type of the intermediate value be γ (i.e., the function ϕ has type $\phi :: \beta \to \gamma$).

The implementation of the reduce_b skeleton consists of the following three steps:

1. local reduction for each segment,
2. gathering local results to the root processor, and
3. global reduction on the root processor.

Step 1. Local Reduction The bottom-up computation of the reduce_b skeleton can be computed by reversed traversal on the array using a stack for the intermediate results. Firstly we apply REDUCE_LOCAL function to each local segment to reduce it to a value. In the computation of the REDUCE_LOCAL function, we need to apply functions ϕ and either ψ_l or ψ_r to the terminal node and its ancestors while we apply function k to the other internal nodes. We apply function k, not ϕ and ψ_n, for reasons of efficiency. To specify where the terminal node or its ancestor is in the stack, we use a variable d that indicates the position. Note that in the computation of the REDUCE_LOCAL function, the stack has at most one node among the terminal node and its ancestors.

```
REDUCE_LOCAL(k, φ, ψ_l, ψ_r, seg)
    stack ← ∅; d ← −∞;
    for i ← seg.size − 1 to 0
        if (isLeaf(seg[i]))
            stack ← seg[i]; d ← d + 1;
        if (isNode(seg[i]))
            lv ← stack; rv ← stack;
            if (d == 0)       stack ← ψ_l(lv, φ(seg[i]), rv);
            else if (d == 1)  stack ← ψ_r(lv, φ(seg[i]), rv); d ← 0;
            else              stack ← k(lv, seg[i], rv); d ← d − 1;
        if (isTerminal(seg[i]))
            stack ← φ(seg[i]); d ← 0;
    top ← stack; return top;
```

In this step, we traverse arrays in the reversed order using a stack, where functions ϕ and either ψ_l or ψ_r is applied to the terminal node and its ancestors and function k is applied to the other internal nodes. Thus, the cost of REDUCE_LOCAL is given as

$$t_1(\text{REDUCE\_LOCAL}) = \left(\frac{L_i}{2} - D_i\right) \times t_1(k) + D_i \times (t_1(\phi) + \max(t_1(\psi_L), t_1(\psi_R))) \ .$$

Step 2. Gathering Local Results to Root Processor In the second step, we gather all the local results to the processors. This is easily done by using MPI's processor-to-processor communication. The communication cost is given by the number of leaf segments and the number of internal segments.

$$t_P(\text{Step 2}) = \frac{M}{2} \times c_\alpha + \frac{M}{2} \times c_\gamma$$

After this step, the gathered values are put in array gt.

Step 3. Global Reduction on Root Processor Finally we compute the result of the reduce_b skeleton by applying REDUCE_GLOBAL function to the array of local results. This computation is performed on the root processors. We can compute the result by applying ψ_n for each internal node in a bottom-up manner and thus we implement the bottom-up computation by a reversed traversal using a stack on the array for the global structure.

```
REDUCE_GLOBAL(ψ_n, gt)
    stack ← ∅;
    for i ← gt.size − 1 to 0
        if (isLeaf(gt[i]))
            stack ← gt[i];
        if (isNode(gt[i]))
            lv ← stack; rv ← stack; stack ← ψ_n(lv, gt[i], rv)
    top ← stack; return top;
```

In this step the function ψ_n is applied to each internal node and thus the cost of REDUCE_GLOBAL is given as follows.

$$t_1(\text{REDUCE\_GLOBAL}) = \frac{M}{2} \times t_1(\psi_n)$$

Summarizing the discussion above, we can give the cost model of the reduce_b skeleton.

$$\begin{aligned} & t_P(\mathsf{reduce}_b) \\ & \quad = \max_p \sum_{pr(i)=p} t_1(\text{REDUCE\_LOCAL}) + t_P(\text{Step 2}) + t_1(\text{REDUCE\_GLOBAL}) \\ & \quad = \max_p \sum_{pr(i)=p} \left(L_i \times \frac{t_1(k)}{2} + D_i \times (-t_1(k) + t_1(\phi) + \max(t_1(\psi_l), t_1(\psi_r))) \right) \\ & \qquad + M \times \frac{c_\alpha + c_\gamma + t_1(\psi_n)}{2} \end{aligned}$$

4.3 Implementation and Cost Model of Upwards Accumulate Skeleton

Next, we develop the implementation of the uAcc_b skeleton called with function k and auxiliary functions $k = \langle \phi, \psi_N, \psi_L, \psi_R \rangle_u$. Let the type of the uAcc_b skeleton be $\mathsf{uAcc}_b :: (\beta \to \alpha \to \alpha \to \alpha) \to \mathsf{BTree}\ \alpha\ \beta \to \mathsf{BTree}\ \alpha\ \alpha$ and the type of intermediate value be γ in the same way as the reduce_b skeleton.

The implementation of the upwards accumulation on distributed trees consists of the following five steps:

1. local upwards accumulation for each segment,
2. gathering results of local reduction to the root processor,
3. global upwards accumulation on the root processor,
4. distributing results of global upwards accumulation, and
5. local update for each internal segment.

Step 1. Local Upwards Accumulation At the first step, we apply the following function uAcc_local to each segment to compute local upwards accumulation. This function puts the intermediate result to array seg' if a node has no terminal node as descendants. (This result value is indeed the result of the uAcc skeleton.) This function returns the result of the local reduction and the array seg'.

```
uAcc_local(k, φ, ψ_l, ψ_r, seg)
    stack ← ∅; d ← −∞;
    for i ← seg.size − 1 to 0
        if (isLeaf(seg[i]))
            seg'[i] ← seg[i]; stack ← seg'[i]; d ← d + 1;
        if (isNode(seg[i]))
            lv ← stack; rv ← stack;
            if (d == 0)       stack ← ψ_l(lv, φ(seg[i]), rv); d ← 0;
            else if (d == 1)  stack ← ψ_r(lv, φ(seg[i]), rv); d ← 0;
            else              seg'[i] ← k(lv, seg[i], rv); stack ← seg'[i]; d ← d − 1;
        if (isTerminal(seg[i]))
            stack ← φ(seg[i]); d ← 0;
    top ← stack; return(top, seg');
```

In the computation of the uAcc_local function, ϕ and either of ψ_l or ψ_r are applied to each node on the path from the critical node to the root, and k is applied to the other internal nodes. Since the number of internal nodes is a half of L_i, we obtain the cost of the uAcc_local function as follows. This cost is the same as that of reduce_local function.

$$t_1(\text{uAcc\_local}) = \left(\frac{L_i}{2} - D_i\right) \times t_1(k) + D_i \times (t_1(\phi) + \max(t_1(\psi_l), t_1(\psi_r)))$$

Step 2. Gathering Results of Local Reduction to Root Processor In the second step, we gather the results of the local reduction to the global structure gt of the root processor. From each leaf segment a value of type α is communicated, and from each internal segment a value of type γ is communicated. Since the number of leaf segments and the number of internal segments are almost $M/2$ respectively, the communication cost of the second step is given as follows.

$$t_P(\text{Step 2}) = \frac{M}{2} \times c_\alpha + \frac{M}{2} \times c_\gamma$$

Step 3. Global Upward Accumulation on Root Processor In the third step, we compute the upwards accumulation for the global structure gt on the root processor. Function uAcc_global performs sequential upwards accumulation using function ψ_n.

```
uAcc_global(ψ_n, gt)
    stack ← ∅;
    for i ← gt.size − 1 to 0
        if (isLeaf(gt[i]))
            gt'[i] ← gt[i];
        if (isNode(gt[i]))
            lv ← stack; rv ← stack; gt'[i] ← ψ_n(lv, gt[i], rv);
        stack ← gt'[i];
    return(gt');
```

In this function, we apply function ψ_n to each internal segment of gt, and thus the cost of the third step is given as

$$t_1(\text{uAcc\_global}) = \frac{M}{2} \times t_1(\psi_n) \ .$$

Step 4. Broadcasting Global Results At the fourth step, we send the results of global upwards accumulation to processors, where two values are sent to each internal segment and no value is sent to each leaf segment. All the values have type α after the global upwards accumulation, and thus the communication cost in the fourth step is given as follows.

$$t_P(\text{Step 4}) = M \times c_\alpha$$

Step 5. Local Update on Path from Root to Terminal Node At the last step, we apply function uAcc_update to each internal segment. The two values pushed to the stack at the beginning of the function are the values passed in the previous step. These two values correspond to the results of children of the terminal node. Note that in the last step we only compute the missing values left in the segment seg'.

```
uAcc_update(k, seg, seg', lc, rc)
    stack ← ∅; stack ← rc; stack ← lc;
    d ← −∞;
    for i ← seg.size − 1 to 0
        if (isLeaf(seg[i]))
            stack ← seg'[i]; d ← d + 1;
        if (isNode(seg[i]))
            lv ← stack; rv ← stack;
            if (d == 0)       seg'[i] ← k(lv, seg[i], rv); stack ← seg'[i];
            else if (d == 1)  seg'[i] ← k(lv, seg[i], rv); stack ← seg'[i]; d ← 0;
            else              stack ← seg'[i]; d ← d − 1;
        if (isTerminal(seg[i]))
            lv ← stack; rv ← stack;
            seg'[i] ← k(lv, seg[i], rv); stack ← seg'[i]; d ← 0;
    return(seg');
```

In this step, function k is applied to the nodes on the path from the terminal node to the root node for each internal segment. Noting that the depth of the terminal nodes is D_i, we can give the cost of uAcc_update as follows.

$$t_1(\text{uAcc\_update}) = D_i \times t_1(k)$$

Summarizing the discussion above we can specify the cost model of the uAcc_b skeleton.

$$\begin{aligned}
&t_P(\mathsf{uAcc}_b) \\
&\quad = \max_p \sum_{pr(i)=p} t_1(\text{uAcc_local}) + t_P(\text{Step 2}) + t_1(\text{uAcc_global}) \\
&\qquad + t_P(\text{Step 4}) + \max_p \sum_{pr(i)=p} t_1(\text{uAcc_update}) \\
&\quad = \max_p \sum_{pr(i)=p} \left(L_i \times \frac{t_1(k)}{2} + D_i \times (t_1(\phi) + \max(t_1(\psi_l), t_1(\psi_r))) \right) \\
&\qquad + M \times (3c_\alpha + c_\gamma + t_1(\psi_n))/2
\end{aligned}$$

4.4 Implementation and Cost Model of Downwards Accumulate Skeleton

Finally we develop the implementation and the cost model for the dAcc_b skeleton called with functions (g_l, g_r) and auxiliary functions $(g_l, g_r) = \langle \phi_l, \phi_r, \psi_u, \psi_d \rangle_d$. Let the type of the skeleton be $\mathsf{dAcc}_b :: (\gamma \to \beta \to \gamma, \gamma \to \beta \to \gamma) \to \gamma \to \mathsf{BTree}\ \alpha\ \beta \to \mathsf{BTree}\ \gamma\ \gamma$ and the type of the intermediate value be δ (i.e., the function ϕ_l has type $\phi_l :: \beta \to \delta$, for example.).

The implementation of the dAcc_b skeleton also consists of the five steps as follows:

1. computing two intermediate values for each internal segment,
2. gathering local results to the root processor,
3. global downwards accumulation on the root processor,
4. distributing the results of global downwards accumulation, and
5. local downwards accumulation for each segment.

Step 1. Computing Local Intermediate Values In the first step, we compute for each internal segment two local intermediate values that are used updating the accumulative parameter from the root node to the both children of the terminal node. To minimize the computation cost we first find the terminal node and then compute two values only on the path from the terminal node to the root node. We implement this computation by the following function dAcc_path, in which the computation is done by a reversed traversal on the array with an integer d instead of a stack.

```
dAcc_path(φ_l, φ_r, ψ_u, seg)
    d ← −∞;
    for i ← seg.size − 1 to 0
        if (isLeaf(seg[i]))
            d ← d + 1;
        if (isNode(seg[i]))
            if (d == 0)
                toL = ψ_u(φ_l(seg[i]), toL); toR = ψ_u(φ_l(seg[i]), toR);
            else if (d == 1)
                toL = ψ_u(φ_r(seg[i]), toL); toR = ψ_u(φ_r(seg[i]), toR);
                d ← 0;
            else
                d ← d − 1;
        if (isTerminal(seg[i]))
            toL ← φ_l(seg[i]); toR ← φ_r(seg[i]);
            d ← 0;
    return (toL, toR);
```

In this step we apply ψ_u and either ϕ_l or ϕ_r twice for each node on the path from the terminal node to the root node. Thus the cost of the dAcc_path function is given as follows.

$$t_1(\text{dAcc\_path}) = D_i \times (\max(t_1(\phi_l), t_1(\phi_r)) + 2t_1(\psi_u))$$

Step 2. Gathering Local Results to Root Processor In the second step, we gather the local results of the internal segments to the root processor. Since the two intermediate values have type δ and the number of internal segments is $M/2$, the communication cost in the second step is given as follows.

$$t_P(\text{Step 2}) = M \times c_\delta$$

The two local results from each internal segment are put to the array of the global tree structure gt.

Step 3. Global Downwards Accumulation In the third step, we compute global downwards accumulation on the root processor. We implement this global downwards accumulation with a forward traversal using a stack as shown in the following function dAcc_global. The initial value of accumulative parameter is pushed to the stack, and then the accumulative parameter in the stack is updated with the local results given in the previous step.

For each segment, the result of global accumulation is the accumulative parameter passed to the root node of the segment.

```
dAcc_global(ψ_d, c, gt)
    stack ← ∅; stack ← c;
    for i ← 0 to gt.size − 1
        if (isLeaf(gt[i]))
            gt'[i] ← stack;
        if (isNode(gt[i]))
            gt'[i] ← stack; (toL, toR) ← gt[i];
            stack ← ψ_d(gt'[i], toR); stack ← ψ_d(gt'[i], toL);
    return gt';
```

The dAcc_global function applies function ψ_d twice for each internal segment in the global structure. Therefore, the computational cost of the dAcc_global function is given as follows.

$$t_1(\text{dAcc\_global}) = M \times t_1(\psi_d)$$

Step 4. Distributing Global Results In the fourth step, we distribute the results of global downwards accumulation to the corresponding processor. Since each result of global downwards accumulation has type γ, the communication cost in the fourth step is given as follows.

$$t_P(\text{step 4}) = M \times c_\gamma$$

Step 5. Local Downwards Accumulation Finally, we compute local downwards accumulation for each segment. The initial value c' of the accumulative parameter is given in the previous step. Note that the definition of the following dAcc_local function is just the same as the sequential version of the downwards accumulation on the serialized array.

```
dAcc_local(g_l, g_r, c', seg)
    stack ← ∅; stack ← c';
    for i ← 0 to seg.size − 1
        if (isLeaf(seg[i]))
            seg'[i] ← stack;
        if (isNode(seg[i]))
            seg'[i] ← stack; stack ← g_r(seg'[i], seg[i]); stack ← g_l(seg'[i], seg[i]);
        if (isTerminal(seg[i]))
            seg'[i] ← stack;
    return seg';
```

The local downwards accumulation applies functions g_l and g_r for each internal node. Since the number of the internal nodes are almost $L_i/2$, the computational cost of the dAcc_local function is given as follows.

$$t_1(\text{dAcc\_local}) = \frac{L_i}{2} \times (t_1(g_l) + t_1(g_r))$$

Summarizing the discussion above, we obtain the following cost model for the dAcc_b skeleton.

$$
\begin{aligned}
& t_P(\mathsf{dAcc}_b) \\
& \quad = \max_p \sum_{pr(i)=p} t_1(\textsc{dAcc\_path}) + t_P(\text{Step 2}) + t_1(\textsc{dAcc\_global}) \\
& \qquad + t_P(\text{Step 4}) + \max_p \sum_{pr(i)=p} t_1(\textsc{dAcc\_local}) \\
& \quad = \max_p \sum_{pr(i)=p} \left(L_i \times \frac{t_1(g_l) + t_1(g_r)}{2} + D_i \times (\max(t_1(\phi_l), t_1(\phi_r)) + 2t_1(\psi_u)) \right) \\
& \qquad + M \times (c_\delta + t_1(\psi_d) + c_\gamma)
\end{aligned}
$$

5 Optimal Division of Binary Trees based on Cost Model

As we stated at the beginning of Section 3, locality and load balance are two major properties in developing efficient parallel programs in particular on distributed-memory parallel computers. By using the m-bridges for dividing and distributing a binary tree, we enjoy good locality with large m, while we enjoy good load balance with smaller m. Therefore, we need to find an appropriate value for m.

First we give the criterion among parameters of the cost model. From Lemma 3 and the representation of local segments in Fig. 5,

$$L_i \leq m \tag{2}$$

holds. Since the maximum height of a tree is a half of the number of nodes, we obtain

$$D_i \leq L_i/2 \leq m/2 \ . \tag{3}$$

From Lemmas 4 and 5, the number of local segments M is bound as

$$\frac{1}{2}\left(\frac{N}{m} - 1\right) \leq M \leq \frac{2N}{m} - 1 \ . \tag{4}$$

We distribute the local segments to processors so as to obtain good load balance. By transforming the cost model using inequality (3), we obtain the following simpler form.

$$
\begin{aligned}
& \max_p \sum_{pr(i)=p} (L_i \times t_l + D_i \times t_d) + M \times t_m \\
& \qquad \leq \max_p \sum_{pr(i)=p} \left(L_i \times t_l + \frac{L_i}{2} \times t_d \right) + M \times t_m \\
& \qquad = (\max_p \sum_{pr(i)=p} L_i) \times \left(t_l + \frac{t_d}{2} \right) + M \times t_m
\end{aligned}
$$

Next we want to bound the maximum of summation $\max_p \sum_{pr(i)=p} L_i$ by the parameter m, N, and P. One easy way to implement the load balancing is distributing the local segments greedily from the largest one. Since the maximum number of nodes in a local segment is m as stated in inequality (2) and the total number of nodes in the original binary tree is N, we can bound the summation as follows:

$$\max_p \sum_{pr(i)=p} L_i \leq \frac{N}{P} + m$$

where P denotes the number of processors. By substituting this inequality to the cost model, we can bound the cost of the worst case.

$$\max_p \sum_{pr(i)=p} (L_i \times t_l + D_i \times t_d) + M \times t_m \leq \left(\frac{N}{P} + m\right) \times \left(t_l + \frac{t_d}{2}\right) + M \times t_m \qquad (5)$$

Now we want to minimize the worst-case cost given in the right-hand side of inequality (5). By substituting the parameter M (inequality (4)), the worst-case cost is bound with respect to m. We can bound the worst-case cost for smaller m as

$$\left(\frac{N}{P} + m\right) \times \left(t_l + \frac{t_d}{2}\right) + M \times t_m \leq \left(\frac{N}{P} + m\right) \times \left(t_l + \frac{t_d}{2}\right) + \frac{1}{2}\left(\frac{N}{m} - 1\right) \times t_m \ ,$$

and we can bound the worst-case cost for larger m as

$$\left(\frac{N}{P} + m\right) \times \left(t_l + \frac{t_d}{2}\right) + M \times t_m \leq \left(\frac{N}{P} + m\right) \times \left(t_l + \frac{t_d}{2}\right) + \frac{2N}{m} - 1 \times t_m \ .$$

From these bounds, we can minimize the worst-case cost for some value m in the following range.

$$\sqrt{\frac{t_m}{2t_l + t_d}}\sqrt{N} \leq m \leq 2\sqrt{\frac{t_m}{2t_l + t_d}}\sqrt{N}$$

This new bound for the parameter m is much smaller than the previous studies [18,31]. In Section 6, we will show several experiment results that support this discussion.

6 Experiment Results

To confirm the efficiency of the implementation of binary-tree skeletons, we made several experiments. We used our PC-cluster of uniform PCs with Pentium 4 2.8 GHz CPU and 2 GByte memory connected with Gigabit Ethernet. The compiler and MPI library used are gcc 4.1.1 and MPICH 1.2.7, respectively.

We used the skeletal parallel program that solves the party planning problem in Fig. 3. The input trees are (1) a balanced tree, (2) a randomly generated tree and (3) a fully ill-balanced tree, each with 16777215 ($= 2^{24} - 1$) nodes. The parameters of the cost model are $t_l = 0.18$ μs, $t_d = 0.25$ μs, and $t_m = 100$ μs on our PC cluster.

Figure 6 shows the the general performance of the tree skeletons. Each execution time excludes the initial data distribution and final gathering. The speedups are plotted against the efficient sequential implementation of the program. As seen in these plots, the implementation shows not only scalability but also good sequential performance. For the fully ill-balanced tree the implementation performs worse but this is caused by the factor of $D_i \times t_d$ ($\sim 0.7 L_i \times t_l$) introduced for parallelism.

To analyze more in detail, we made more experiments by changing the value of m. The results are shown in Fig. 7. Roughly speaking, as seen from Fig. 7 (left), the implementation of tree accumulations scales under both large and small m. Figure 7 (right) plots the execution time with respect to the parameter m. The performance gets worse for too small m or too large m, where good performance is shown under the range $5 \times 10^4 < m < 1 \times 10^5$ computed from inequality (5) with substitution of the parameters t_l, t_d, t_m, and N given above.

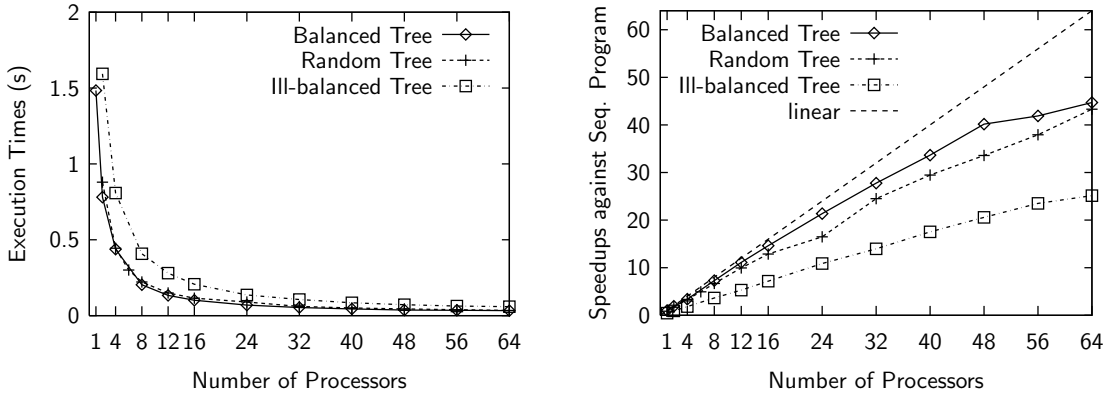


Fig. 6. Execution times and speedups against sequential program where $m = 2 \times 10^4$.

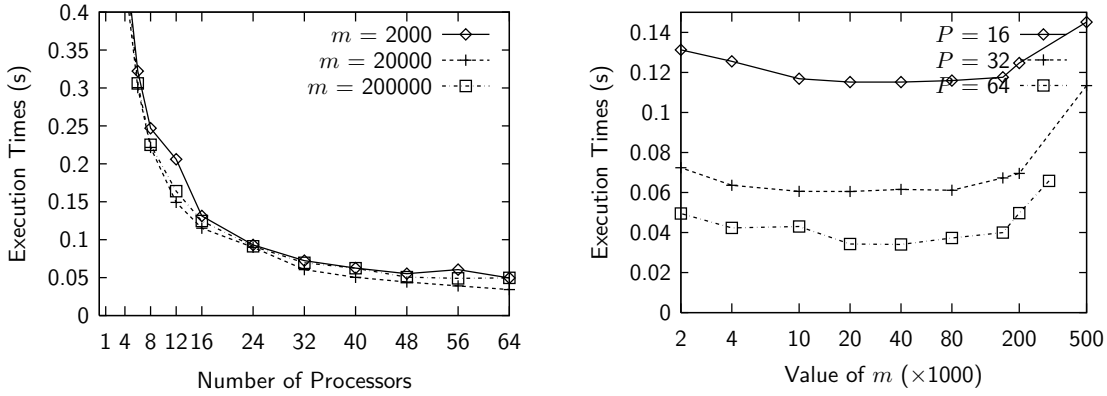


Fig. 7. Execution times changing parameter m for the randomly generated tree.

7 Related Work

Tree contraction algorithms, whose idea was first proposed by Miller and Reif [25], are very important parallel algorithms for efficient manipulations of trees. Many researchers have devoted themselves to developing efficient implementations of the tree contraction algorithms on various parallel models [1–3, 6, 8, 11, 15, 23, 24, 38]. Among them, Gibbons and Rytter developed an cost-optimal algorithm on CREW PRAM [11]; 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]; and recently more efficient implementations are discussed [2, 38] for symmetric multiprocessors (SMP) and chip-level multiprocessing (CMP). A lot of tree programs have been described by the tree contraction algorithms [3, 6, 11, 14, 17, 26–29].

There have been several studies on the implementations of parallel tree skeletons [12, 13, 16, 18, 33–35]. Gibbons et al. [13, 34] have developed an implementation of parallel tree skeletons based on the tree contraction algorithms. There algorithm can be used on many parallel computers, due to the various implementation algorithms on various parallel computers. Skillicorn [35] and our previous paper [18] have discussed implementations of parallel tree skeletons based on the division of trees. Compared with these implementation algorithms, our implementation is unique in terms of data structure of local segments for better sequential performance and the cost model supporting good division of trees. As far as we are aware, we are the first who implement the parallel tree skeletons as a parallel skeleton library. Our implementation of the tree skeletons will be available as a part of SkeTo library [22]. In terms of manipulations of general trees, which are formalized as parallel

rose-tree skeletons [20], some of them are implemented efficiently in parallel [16, 33]. Sevilgen et al. [33] has shown an implementation algorithm for tree accumulations on general trees where rather strict conditions are requested for efficient implementation. Kakehi et al. [16] has developed an efficient implementation of tree reduction on general trees based on the serialized representation like XML formats.

8 Conclusion

In this paper, we have developed an efficient implementation of parallel tree skeletons. Not only our implementation shows good performance even against sequential programs, but also the cost model of the implementation helps us to divide a tree into segments with good load balance. The implementation will be available as a part of SkeTo library[1]. One of our future work is to develop a profiling system that determines more accurate parameter m for dividing trees.

References

1. Karl R. Abrahamson, N. Dadoun, David G. Kirkpatrick, and Teresa M. Przytycka. A simple parallel tree contraction algorithm. *Journal of Algorithms*, 10(2):287–302, June 1989.
2. David A. Bader, Sukanya Sreshta, and Nina R. Weisse-Bernstein. Evaluating arithmetic expressions using tree contraction: A fast and scalable parallel implementation for symmetric multiprocessors (smps) (extended abstract). In Sartaj Sahni, Viktor K. Prasanna, and Uday Shukla, editors, *High Performance Computing – HiPC 2002, 9th International Conference, Bangalore, India, December 18–21, 2002, Proceedings*, volume 2552 of *Lecture Notes in Computer Science*, pages 63–78. Springer, 2002.
3. Raja P. K. Banerjee, Vineet Goel, and Amar Mukherjee. Efficient parallel evaluation of CSG tree using fixed number of processors. In *ACM Symposium on Solid Modeling Foundations and CAD/CAM Applications, May 19–21, 1993, Montreal, Canada*, pages 137–146, 1993.
4. Richard S. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall Series in Computer Science. Prentice Hall, 2nd edition, April 1998.
5. Richard S. Bird. Maximum marking problems. *Journal of Functional Programming*, 11(4):411–424, July 2001.
6. Richard Cole and Uzi Vishkin. The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time. *Algorithmica*, 3:329–346, March 1988.
7. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, second edition, Sptember 2001.
8. Frank K. H. A. Dehne, Afonso Ferreira, Edson Cáceres, Siang W. Song, and Alessandro Roncato. Efficient parallel graph algorithms for coarse-grained multicomputers and BSP. *Algorithmica*, 33(2):183–200, January 2002.
9. Hossain Deldari, John R. Davy, and Peter M. Dew. Parallel CSG, skeletons and performance modeling. In *Proceedings of the Second Annual CSI Computer Conference (CSICC'96)*, pages 115–122, 1996.
10. Krzysztof Diks and Torben Hagerup. More general parallel tree contraction: Register allocation and broadcasting in a tree. *Theorecital Computer Science*, 203(1):3–29, August 1998.
11. Alan Gibbons and Wojciech Rytter. An optimal parallel algorithm for dynamic expression evaluation and its applications. In Kesav V. Nori, editor, *Foundations of Software Technology and Theoretical Computer Science, Sixth Conference, New Delhi, India, December 18–20, 1986, Proceedings*, volume 241 of *Lecture Notes in Computer Science*, pages 453–469. Springer, 1986.

[1] `http://www.ipl.t.u-tokyo.ac.jp/sketo/`

12. Jeremy Gibbons. Computing downwards accumulations on trees quickly. In Gopal Gupta, George Mohay, and Rodney Topor, editors, *Proceedings of the 16th Australian Computer Science Conference*, pages 685–691, 1993.

13. Jeremy Gibbons, Wentong Cai, and David B. Skillicorn. Efficient parallel algorithms for tree accumulations. *Science of Computer Programming*, 23(1):1–18, October 1994.

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

15. Gustedt Jens. Communication and memory optimized tree contraction and list ranking. Technical report, INRIA, Unité de recherche, Rhône-Alpes, Montbonnot-Saint-Martin, FRANCE, December 2000.

16. Kazuhiko Kakehi, Kiminori Matsuzaki, Kento Emoto, and Zhenjiang Hu. An practicable framework for tree reductions under distributed memory environments. Technical Report METR 2006-64, Department of Mathematical Informatics, Graduate School of Information Science and Technology, University of Tokyo, December 2006.

17. Kiminori Matsuzaki, Zhenjiang Hu, Kazuhiko Kakehi, and Masato Takeichi. Systematic derivation of tree contraction algorithms. *Parallel Processing Letters*, 15(3):321–336, September 2005.

18. Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Implementation of parallel tree skeletons on distributed systems. In *The Third Asian Workshop on Programming Languages and Systems, APLAS'02, Shanghai Jiao Tong University, Shanghai, China, November 29 – December 1, 2002, Proceedings*, pages 258–271, 2002.

19. Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Parallelization with tree skeletons. volume 2790 of *Lecture Notes in Computer Science*, pages 789–798. Springer, 2003.

20. Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Parallel skeletons for manipulating general trees. *Parallel Computing*, 32(7–8):590–603, September 2006.

21. Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Towards automatic parallelization of tree reductions in dynamic programming. In Phillip B. Gibbons and Uzi Vishkin, editors, *SPAA 2006: Proceedings of the 18th Annual ACM Symposium on Parallelism in Algorithms and Architectures, July 30–August 2, 2006, Cambridge, Massachusetts, USA*, pages 39–48. ACM Press, 2006.

22. Kiminori Matsuzaki, Hideya Iwasaki, Kento Emoto, and Zhenjiang Hu. A library of constructive skeletons for sequential style of parallel programming. In *InfoScale '06: Proceedings of the 1st international conference on Scalable information systems*, volume 152 of *ACM International Conference Proceeding Series*, page 13. ACM Press, 2006.

23. Ernst W. Mayr and Ralph Werchner. Optimal routing of parentheses on the hypercube. *Journal of Parallel and Distbributed Computing*, 26(2):181–192, April 1995.

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

25. Gary L. Miller and John H. Reif. Parallel tree contraction and its application. In *26th Annual Symposium on Foundations of Computer Science, 21–23 October 1985, Portland, Oregon, USA*, pages 478–489. IEEE Computer Society, 1985.

26. Gary L. Miller and John H. Reif. Parallel tree contraction, part 2: Further applications. *SIAM Journal on Computing*, 20(6):1128–1147, 1991.

27. Gary L. Miller and Shang-Hua Teng. Dynamic parallel complexity of computational circuits. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing, 25–27 May 1987, New York City, NY, USA*, pages 254–263. ACM Press, 1987.

28. Gary L. Miller and Shang-Hua Teng. Tree-based parallel algorithm design. *Algorithmica*, 19(4):369–389, December 1997.

29. Gary L. Miller and Shang-Hua Teng. The dynamic parallel complexity of computational circuits. *SIAM Journal on Computing*, 28(5):1664–1688, 1999.

30. S. Peyton Jones and J. Hughes. Report on the programming language Haskell 98: A non-strict, purely functional language. Available from `http://www.haskell.org/`, February 1999.

31. John H. Reif, editor. *Synthesis of Parallel Algorithms*. Morgan Kaufmann Publishers, February 1993.

32. Isao Sasano, Zhenjiang Hu, and Masato Takeichi. Generation of efficient programs for solving maximum multi-marking problems. In Walid Taha, editor, *Semantics, Applications, and Implementation of Program Generation, Second International Workshop, SAIG 2001, Florence, Italy, September 6, 2001, Proceedings*, volume 2196 of *Lecture Notes in Computer Science*, pages 72–91. Springer, 2001.

33. Fatih E. Sevilgen, Srinivas Aluru, and Natsuhiko Futamura. Parallel algorithms for tree accumulations. *Journal of Parallel and Distbributed Computing*, 65(1):85–93, 2005.

34. David B. Skillicorn. *Foundations of Parallel Programming*, volume 6 of *Cambridge International Series on Parallel Computation*. Cambridge University Press, 1994.

35. David B. Skillicorn. Parallel implementation of tree skeletons. *Journal of Parallel and Distbributed Computing*, 39(2):115–125, December 1996.

36. David B. Skillicorn. A parallel tree difference algorithm. *Information Processing Letters*, 60(5):231–235, December 1996.

37. David B. Skillicorn. Structured parallel computation in structured documents. *Journal of Universal Computer Science*, 3(1):42–68, January 1997.

38. Uzi Vishkin. A no-busy-wait balanced tree parallel algorithmic paradigm. In Gary Miller and Shang-Hua Teng, editors, *SPAA 2000: Proceedings of the 12th Annual ACM Symposium on Parallel Algorithms and Architectures, July 09–13, 2000, Bar Harbor, Maine, USA*, pages 147–155. ACM Press, 2000.