DOI 10.12694/scpe.v18i1.1231
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 18, Number 1, pp. 17–34. http://www.scpe.org
© 2017 SCPE

EFFICIENT IMPLEMENTATION OF TREE SKELETONS ON DISTRIBUTED-MEMORY PARALLEL COMPUTERS*

KIMINORI MATSUZAKI†

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

Key words: Parallel skeletons, Tree, Cost model, Parallel programming

AMS subject classifications. 68N19, 68W10

1. Introduction.

Parallel tree skeletons, first formalized by Skillicorn [33, 34], are basic computational patterns of parallel programs manipulating trees. By using parallel tree skeletons, we can develop parallel programs without considering low-level parallel implementation and details of parallel computers. There have been several studies on the systematic methods of developing parallel programs by means of parallel tree skeletons [6, 19, 21, 35, 36].

For efficient parallel tree manipulations, tree contraction algorithms have been intensively studied [1, 5, 24, 25, 38]. Many tree contraction algorithms were given on various parallel computational models, for instance, EREW PRAM [1], Hypercubes [24], and BSP/CGM [5]. Several parallel tree manipulations were developed based on the tree contraction algorithms [1, 7]. For tree skeletons, Gibbons et al. [11] developed an implementation algorithm based on tree contraction algorithms.

In this paper, we propose an efficient implementation of parallel tree skeletons for binary trees on distributed-memory parallel computers. Compared with the implementations so far, our implementation has three new features.

First, it has less overheads of parallelism. Locality is one of the most important properties in developing efficient parallel programs especially for distributed-memory computers. We adopt the technique of m-bridges [8, 30] in the basic graph theory to divide binary trees into segments with high locality.

Second, it has high sequential performance. The performance of sequential computation parts is as important as that of the communication parts. We represent a local segment as a serialized array and implement local computation in tree skeletons with loops rather than recursive functions.

Third, it has a cost model. We formalize a cost model of our parallel implementation. The cost model helps us to divide binary trees with good load balance.

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

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

2. Parallel Tree Skeletons.

*An earlier version of this paper appeared in PAPP2007, part of ICCS2007[16]. This work was partially supported by Japan Society for the Promotion of Science, Grant-in-Aid for Scientific Research (B) No. 17300005, and the Ministry of Education, Culture, Sports, Science and Technology, Grant-in-Aid for Young Scientists (B) No. 18700021.

†School of Information, Kochi University of Techonology, 185 Tosayamadacho-Miyanokuchi, Kami, Kochi 782–8502 Japan. (matsuzaki.kiminori@kochi-tech.ac.jp)

$$
\begin{array}{l}
\mathsf{map} :: (\alpha \to \gamma, \beta \to \delta, BTree\langle \alpha, \beta \rangle) \to BTree\langle \gamma, \delta \rangle \\
\mathsf{map}(k_l, k_n, BLeaf(a)) \qquad = BLeaf(k_l(a)) \\
\mathsf{map}(k_l, k_n, BNode(l, b, r)) = BNode(\mathsf{map}(k_l, k_n, l), k_n(b), \mathsf{map}(k_l, k_n, r)) \\
\\
\mathsf{reduce} :: ((\alpha, \beta, \alpha) \to \alpha, BTree\langle \alpha, \beta \rangle) \to \alpha \\
\mathsf{reduce}(k, BLeaf(a)) \qquad = a \\
\mathsf{reduce}(k, BNode(l, b, r)) = k(\mathsf{reduce}(k, l), b, \mathsf{reduce}(k, r)) \\
\\
\mathsf{uAcc} :: ((\alpha, \beta, \alpha) \to \alpha, BTree\langle \alpha, \beta \rangle) \to BTree\langle \alpha, \alpha \rangle \\
\mathsf{uAcc}(k, BLeaf(a)) \qquad = BLeaf(a) \\
\mathsf{uAcc}(k, BNode(l, b, r)) = \mathbf{let}\ b' = \mathsf{reduce}(k, BNode(l, b, r)) \\
\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ BNode(\mathsf{uAcc}(k, l), b', \mathsf{uAcc}(k, r)) \\
\\
\mathsf{dAcc} :: ((\gamma, \beta) \to \gamma, (\gamma, \beta) \to \gamma, \gamma, BTree\langle \alpha, \beta \rangle) \to BTree\langle \gamma, \gamma \rangle \\
\mathsf{dAcc}(g_l, g_r, c, BLeaf(a)) \qquad = BLeaf(c) \\
\mathsf{dAcc}(g_l, g_r, c, BNode(l, b, r)) = \mathbf{let}\ l' = \mathsf{dAcc}(g_l, g_r, g_l(c, b), l) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad r' = \mathsf{dAcc}(g_l, g_r, g_r(c, b), r) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ BNode(l', c, r')
\end{array}
$$

FIG. 2.1. *Definition of parallel tree skeletons.*

2.1. 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 leaves have values of type α and internal nodes have values of type β is defined as follows.

$$
\begin{array}{l}
\text{data } BTree\langle \alpha, \beta \rangle = BLeaf(\alpha) \\
\qquad\qquad\qquad\qquad \mid BNode(BTree\langle \alpha, \beta \rangle, \beta, BTree\langle \alpha, \beta \rangle)
\end{array}
$$

Functions that manipulate binary trees can be defined by pattern matching. For example, function *root* that returns the value of the root node is as follows.

$$
\begin{array}{l}
root :: BTree\langle \alpha, \alpha \rangle \to \alpha \\
root(BLeaf(a)) \qquad = a \\
root(BNode(l, b, r)) = b
\end{array}
$$

2.2. Parallel Tree Skeletons. Parallel (binary-)tree skeletons are basic computational patterns manipulating binary trees in parallel. In this section, we introduce a set of basic tree skeletons proposed by Skillicorn [33, 34] with minor modifications for later discussion of their implementation (Fig. 2.1).

The tree skeleton map 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. Though there are tree skeletons called zip or zipwith that take multiple trees of the same shape, we omit discussing them since computation of them is almost the same as that of the map skeleton.

The parallel skeleton reduce 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 (upwards accumulate) is a shape-preserving manipulation, which also takes a function k and a binary tree and computes (reduce k) for each subtree.

The parallel skeleton dAcc (downwards accumulate) is another shape-preserving manipulation. 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.

Since a straightforward divide-and-conquer computation according to the definition in Fig. 2.1 has computational cost linear to the height of the tree, it may be inefficient if the input tree is ill-balanced. To guarantee existence of efficient parallel implementations, we impose some conditions on the parameter functions of tree skeletons. The map skeleton requires no condition. For the reduce, uAcc and dAcc skeletons, we formalize the conditions for parallel implementations as existence of auxiliary functions satisfying a certain closure property.

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

$$\text{(2.1)} \quad \begin{aligned} &\text{a: } k(l,b,r) = \psi_n(l,\phi(b),r) \\ &\text{b: } \psi_n(\psi_n(x,l,y),b,r) = \psi_n(x,\psi_l(l,b,r),y) \\ &\text{c: } \psi_n(l,b,\psi_n(x,r,y)) = \psi_n(x,\psi_r(l,b,r),y) \end{aligned}$$

Equations (2.1.b) and (2.1.c) represent the closure property that functions ψ_n, ψ_l, and ψ_r should satisfy. Function ϕ lifts the computation of function k to the domain with closure property as in Equation (2.1.a). We denote the function k satisfying the condition as $k = \langle \phi, \psi_n, \psi_l, \psi_r \rangle_u$.

The dAcc skeleton called 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.

$$\text{(2.2)} \quad \begin{aligned} &\text{a: } g_l(c,b) = \psi_d(c,\phi_l(b)) \\ &\text{b: } g_r(c,b) = \psi_d\ c,\phi_r(b)) \\ &\text{c: } \psi_d(\psi_d(c,b),b') = \psi_d(c,\psi_u(b,b')) \end{aligned}$$

Equation (2.2.c) shows the closure property that functions ψ_d and ψ_u should satisfy. As shown in Equations (2.2.a) and (2.2.b), computations of g_l and g_r are lifted up to the domain with closure property by functions ϕ_l and ϕ_r, respectively. 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 see how we can develop parallel programs with these parallel tree skeletons, we consider the following two problems.

Sum of Values. Consider computing the sum of node values of a binary tree. We can define function *sum* for this problem simply by using the reduce skeleton.

$$\begin{aligned} &\mathit{sum}\ t = \mathsf{reduce}(\mathit{add3}, t) \\ &\qquad\qquad \textbf{where } \mathit{add3}(l,b,r) = l + b + r \end{aligned}$$

In this case, since the operator used is only the associative operator $+$, we have $\mathit{add3} = \langle \mathit{id}, \mathit{add3}, \mathit{add3}, \mathit{add3} \rangle_u$, where *id* is the identity function.

Prefix Numbering. Consider numbering the nodes of a binary tree in the prefix traversing order. We can define function *prefix* for this problem by using the tree skeletons map, uAcc, and dAcc as follows. Note that the result of the uAcc skeleton is a pair of number of nodes of left subtree and number of nodes for each node.

$$\begin{aligned} \mathit{prefix}\ t = &\ \textbf{let } t' = \mathsf{uAcc}(k, \mathsf{map}(f, \mathit{id}, t)) \\ &\ \textbf{in } \mathsf{dAcc}(g_l, g_r, 0, t') \\ &\ \textbf{where } f(a) = (0,1) \\ &\qquad k((l_l,l_s),b,(r_l,r_s)) = (l_s, l_s + 1 + r_s) \\ &\qquad g_l(c,(b_l,b_s)) = c + 1 \\ &\qquad g_r(c,(b_l,b_s)) = c + b_l + 1 \end{aligned}$$

Similar to the case of *sum*, auxiliary functions for the functions g_l and g_r are simply given as $(g_l, g_r) = \langle \lambda(b_l,b_s).1, \lambda(b_l,b_s).b_l + 1, +, + \rangle_d$. For function k, auxiliary functions are given as follows by applying the derivation technique discussed in [21].

$$\begin{aligned} &k = \langle \phi, \psi_n, \psi_l, \psi_r \rangle \\ &\textbf{where } \phi(b) = (1,0,0,1) \\ &\qquad \psi_n((l_l,l_s),(b_0,b_1,b_2,b_3),(r_l,r_s)) \\ &\qquad\quad = (b_0 \times l_s + b_1 \times (l_s + 1 + r_s) + b_2,\ l_s + 1 + r_s + b_3) \\ &\qquad \psi_l((l_0,l_1,l_2,l_3),(b_0,b_1,b_2,b_3),(r_l,r_s)) \\ &\qquad\quad = (0,\ b_0 + b_1,\ (b_0 + b_1) \times l_3 + b_1 \times (1 + r_s) + b_2,\ l_3 + 1 + r_s + b_3) \\ &\qquad \psi_r((l_l,l_s),(b_0,b_1,b_2,b_3,b),(r_0,r_1,r_2,r_3,r)) \\ &\qquad\quad = (0,\ b_1,\ b_1 \times r_3 + b_0 \times l_s + b_1 \times (1 + l_s) + b_2,\ r_3 + 1 + l_s + b_3) \end{aligned}$$

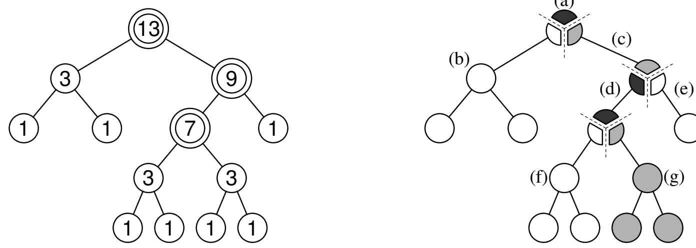

Fig. 3.1. *An example of m-critical nodes and m-bridges. Left: In this binary tree, there are three* 4*-critical nodes denoted by the doubly-lined circles. The number in each node denotes the number of nodes in the subtree. Right: For the same tree there are seven* 4*-bridges, (a)–(g), each of which is a set of connected nodes.*

It is worth noting that the set of auxiliary functions is not unique. For example, we can define another set of auxiliary functions in which an intermediate value has a flag and two values.

In general, auxiliary functions are more complicated than the original function as we have seen in this example. This complexity of auxiliary functions would introduce overheads in the derived parallel algorithms.

3. Division of Binary Trees with High Locality. To develop efficient parallel programs on distributed-memory parallel computers, we need to divide data into smaller parts and distribute them to the processors. Here, the division of data should have the following two properties for efficiency of parallel programs. The first property is *locality.* The data distributed to each processor should be adjacent. If two elements adjacent in the original data are distributed to different processors, then we often need communications between the processors. The second property is *load balance.* The number of elements distributed to each processor should be nearly equal since the cost of local computation is often proportional to the number of elements.

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 each having N/P elements. It is, however, difficult to divide a tree satisfying the two properties due to the nonlinear and irregular structure of binary trees.

In this section, we introduce a division of binary trees based on the basic graph theory [8, 30], and show how to represent the distributed tree structures for efficient implementation of tree skeletons.

3.1. Graph-Theoretic Results for Division of Binary Trees. We start by introducing some graph-theoretic results [8, 30]. Let $size(v)$ denote the number of nodes in the subtree rooted at node v.

Definition 3.1 (m-Critical Node). *Let m be an integer. A node v is called an m-*critical node*, if*

- *v is an internal node, and*
- *for each child v' of v inequality $\lceil size(v)/m \rceil > \lceil size(v')/m \rceil$ holds.*

The m-critical nodes divide a tree into sets of adjacent nodes (m-bridges) as shown in Fig. 3.1.

Definition 3.2 (m-Bridge). *Let m be an integer. An m-*bridge *is a set of adjacent nodes divided by m-critical nodes, that is, a largest set of adjacent nodes in which m-critical nodes are only at the root or bottom.*

In the following of this paper, we assume that each local segment given by dividing a tree is an m-bridge. The global structure of m-bridges also forms a binary tree.

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

Lemma 3.3. *If v_1 and v_2 are m-critical nodes then their least common ancestor is also an m-critical node.*

Lemma 3.4. *If B is an m-bridge of a tree then B has at most one m-critical node among the leaves of it.*

The root node in each m-bridge, except the m-bridge that includes the global root node, is an m-critical node. If we remove the root m-critical node if it exists, it follows from Lemma 3.4 and Definition 3.2 that the m-bridge has at most one m-critical node at its bottom.

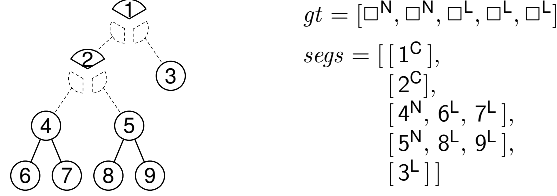

FIG. 3.2. *Array representation of divided binary trees. Each local segment of segs is assigned to one of processors and is not shared. Labels* L*,* N *and* C *denote a leaf, a normal internal node, and an m-critical node, respectively. Each m-critical node is included in the parent segment.*

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 first two lemmas hold on general trees while the last lemma only holds on binary trees.

LEMMA 3.5. *The number of nodes in an m-bridge is at most $m + 1$.*

LEMMA 3.6. *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 3.7. *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 $(N/m - 1)/2$.*

Let N be the number of nodes and P be the number of processors. In the previous studies [8, 18, 30], we divided a tree into m-bridges using the parameter m given by $m = 2N/P$. Under this division we obtain at most $(2P - 1)$ m-bridges and thus each processor handles at most two m-bridges. Of course this division enjoys high locality, but it has poor load balance since the maximum number of nodes passed to a processor may be $2N/P$, which is twice of that for the best lead-balancing case.

In Sect. 5, we will adjust the value m for better division of binary trees based on the cost model of tree skeletons. The idea is to 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 small.

3.2. Data Structure for Distributed Segments. The performance of the sequential computation parts is as important as that of the communication parts.

Generally speaking, tree structures are often implemented using pointers or references. There are, however, two problems in this implementation for large-scale tree applications. First, much memory is required for pointers. Considering trees of integers or real numbers, for example, we can see that the pointers use as much memory as the values do. Furthermore, if we allocate nodes one by one, more memory is consumed to enable each of them to be deallocated. Second, locality is often lost. 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 increases.

To resolve these problems, we represent a binary tree with arrays. We represent a tree divided by the m-bridges using one array *gt* for the global structure and one array of arrays *segs* for the local segments, each of which is given by serializing the tree in the order of prefix traversal. Note that arrays in *segs* are distributed among processors, while each local segment exists in only one processor. Figure 3.2 illustrates the array representation of a distributed tree. Since adjoining elements are aligned one next to another in this representation, we can reduce cache misses.

We introduce some notations for the discussion of implementation algorithms in the next section. Some values may be attached to the global structure, and we write $gt[i]$ to access to the value attached to ith element of global structure. If ith segment in *segs* is distributed on pth processor, we denote $pr(i) = p$. For a given serialized array for a segment *seg*, we use $seg[i]$ to denote the ith value in the serialized array, and use isLeaf($seg[i]$), isNode($seg[i]$) and isCritical($seg[i]$) to check whether the ith node is a leaf, an internal node, and an m-critical node, respectively. Function isRoot(p) checks if the processor p is the root processor or not.

TABLE 4.1
Parameters of 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 m-critical node in the ith segment
c_α	the time needed for communicating one data of type α
$\lvert\alpha\rvert$	the size of a value of type α

4. Implementation and Cost Model of Tree Skeletons. In this section, we show an implementation and a cost model of the tree skeletons on distributed-memory parallel computers. We implement the local computation in tree skeletons using loops and stacks on the serialized arrays, which play an important role in reducing the cache misses and achieving high performance in the sequential computation parts.

We define several parameters for discussion of the cost model (Table 4.1). We assume a homogeneous distributed-memory environments as the computation environment. The computational time of function f executed with p processors is denoted by $t_p(f)$. In particular, $t_1(f)$ denotes the cost of sequential computation of 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 size of a value of type α is denoted by $|\alpha|$.

The cost of tree skeletons can be uniformly given as the sum of the maximum local-computation cost and the global-computation cost as follows.

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

In the expression of the cost, $\sum_{pr(i)=p}$ denotes the summation of cost for local segments associated to processor p. The parameter t_l indicates the cost of computation that is applied to each node in a segment, the parameter t_d indicates the cost of computation that is applied to the nodes on the path from the root to the m-critical node, and the parameter t_m denotes the communication cost required for each segment.

In fact, we can extend the implementation of tree skeletons and the cost model to the bulk-synchronous parallel (BSP) model [37]. The cost of a BSP algorithm is given by the sum of costs of supersteps, which consists of local computation followed by communication and barrier synchronization. The cost of a superstep is given by an expression of the form $w + hg + l$ where w is the maximum cost of local computation, h is the size of messages, g is the cost of communicating a message of size one, and l is the cost of the barrier synchronization. Note that for the parameter g we have $c_\alpha = |\alpha|g$ for any type α.

4.1. Map. Since there is no dependency among nodes in the computation of the map skeleton, we can implement the map skeleton by applying function MAP_LOCAL to each local segment, where the MAP_LOCAL function applies function k_l to each leaf and function k_n to each internal node and the m-critical node in a local segment. The implementation of the map skeleton is given in Fig. 4.1.

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 m-critical node is at most $L_i/2+1$. Ignoring small constants we can specify the computational cost of the MAP_LOCAL function as

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

```
map(k_l, k_n, (gt, segs))
  for i ← 0 to gt.size − 1: begin
    if pr(i) == p then segs'[i] ← MAP_LOCAL(k_l, k_n, segs[i]); endif
  end
  return (gt, segs');

MAP_LOCAL(k_l, k_n, seg)
  for i ← 0 to seg.size − 1: begin
    if isLeaf(seg[i])     then seg'[i] ← k_l(seg[i]); endif
    if isNode(seg[i])     then seg'[i] ← k_n(seg[i]); endif
    if isCritical(seg[i]) then seg'[i] ← k_n(seg[i]); endif
  end
  return seg';
```

FIG. 4.1. *Implementation of* map *skeleton.*

The cost of the map skeleton is as follows.

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

On the BSP model, we can implement the map skeleton with a single superstep without communication. Thus, the BSP cost is given as follows.

$$t_P^{\langle \mathrm{BSP} \rangle}(\mathsf{map}(k_l, k_n, t)) = \max_p \sum_{pr(i)=p} L_i \times \frac{t_1(k_l) + t_1(k_n)}{2} + l$$

4.2. Reduce. We then show an implementation and its cost of the reduce skeleton called with function k and auxiliary functions $k = \langle \phi, \psi_n, \psi_l, \psi_r \rangle_u$. Let the input binary tree have type $BTree\langle \alpha, \beta \rangle$ and intermediate values for auxiliary functions have type γ. The implementation of the reduce skeleton is shown in Fig. 4.2.

Step 1. Local Reduction. The bottom-up computation of the reduce skeleton can be computed by a traversal on the array from right to left 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 REDUCE_LOCAL function,

- we apply functions ϕ and either ψ_l or ψ_r to the m-critical node and its ancestors, and
- we apply function k to the other internal nodes.

Here, applying the function k is cheaper than applying function ϕ and ψ_n, even though $k(l, n, r) = \psi_n(l, \phi(n), r)$ holds with respect to the results of functions. To specify where the m-critical 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, at most one element in the stack has the value of the m-critical node or its ancestors.

In this step, functions ϕ and either ψ_l or ψ_r are applied to the m-critical node and its ancestors (D_i nodes) and function k is applied to the other internal nodes ($(M_i/2 - D_i)$ nodes). Thus, the cost of REDUCE_LOCAL is given as follows.

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

Step 2. Gathering Local Results to Root Processor. In the second step, we gather the local results to the root processor. The communication cost is given by the number of leaf segments of type α and the number of internal segments of type γ.

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

Let the gathered values be put in array gt on the root processor after this step.

```
reduce(k, (gt, segs)) where k = ⟨φ, ψ_n, ψ_l, ψ_r⟩_u
  for i ← 0 to gt.size − 1: begin
    if pr(i) == p then gt[i] ← REDUCE_LOCAL(k, φ, ψ_l, ψ_r, segs[i]); endif
  end
  gather_to_root(gt);
  if isRoot(p) then return REDUCE_GLOBAL(ψ_n, gt); endif

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

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

FIG. 4.2. *Implementation of* reduce *skeleton*

Step 3. Global Reduction on Root Processor. Finally, we compute the result of the reduce skeleton by applying the REDUCE_GLOBAL function to the array of local results. This computation is performed on the root processor. We compute the result by applying ψ_n for each internal node in a bottom-up manner, which is implemented by a traversal with a stack on the array of the global structure from right to left.

In this step the function ψ_n is applied to each internal segment and the cost of REDUCE_GLOBAL is

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

In summary, the cost of the reduce skeleton is given as follows.

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

On the BSP model, we can implement the reduce skeleton with two supersteps: one consists of Steps 1 and

2; the other consists of Step 3. Thus, the BSP cost is given as follows.

$$\begin{aligned} &t_P^{\langle\mathrm{BSP}\rangle}(\mathsf{reduce}\ k) \\ &\quad = \max_p \sum_{pr(i)=p} \left(L_i \times \frac{t_1(k)}{2} + D_i \times (-t_1(k) + t_1(\phi) + \max(t_1(\psi_l), t_1(\psi_r))) \right) \\ &\qquad + M \times t_1(\psi_n)/2 + M \times \frac{|\alpha| + |\gamma|}{2} \times g + 2l \end{aligned}$$

4.3. Upwards Accumulate. Next, we develop an implementation of the uAcc skeleton called with function k and auxiliary functions $k = \langle \phi, \psi_n, \psi_l, \psi_r \rangle_u$. Similar to the reduce skeleton, let the type of input binary tree be $BTree\langle \alpha, \beta \rangle$ and the type of intermediate values be γ. The implementation of the uAcc skeleton is shown in Fig. 4.3.

Step 1. Local Upwards Accumulation. In the first step, we apply function uAcc_local to each segment and compute local upwards accumulation and reduction. This function puts the intermediate results to array seg' if a node has no m-critical node as descendants, since the result value is indeed the result of the uAcc skeleton. This function puts nothing to array seg' if a node is either the m-critical node or an ancestor of the m-critical node. Returned values are the result of local reduction and the array seg'.

In the computation of the uAcc_local function, ϕ and either of ψ_l or ψ_r are applied to each node of the m-critical node and its ancestors (D_i nodes), and k is applied to the other internal nodes ($(L_i/2 - D_i)$ nodes). We obtain the cost of the uAcc_local function as

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

Note that this cost is the same as that of reduce_local function.

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

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

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

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

Step 4. Distributing Global Result. In the fourth step, we send the result of global upwards accumulation to processors, where two values are sent to each internal segment and no values are sent to each leaf segment. Since all the values have type α after the global upwards accumulation, the communication cost of the fourth step is given as

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

Step 5. Local Updates for Each Internal Segment. In the last step, we apply function uAcc_update to each internal segment. At the beginning of the function, the two values pushed in the previous step are pushed to the stack. These two values correspond to the results of children of the m-critical node. Note that in the last step we only compute the missing values in the segment seg', which is given in the local upwards accumulation (Step 1).

```
uAcc(k, (gt, segs)) where k = ⟨φ, ψ_n, ψ_l, ψ_r⟩_u
  for i ← 0 to gt.size − 1: begin
    if pr(i) == p then (gt[i], segs′[i]) ← uAcc_local(k, φ, ψ_l, ψ_r, segs[i]); endif
  end
  gather_to_root(gt)
  if isRoot(p) then gt′ ← uAcc_global(ψ_n, gt); endif
  distribute_from_root(gt′)
  for i ← 0 to gt.size − 1: begin
    if pr(i) == p ∧ isNode(gt′[i]) then
      segs′[i] ← uAcc_update(k, segs[i], segs′[i], gt′[i]) endif
  end
  return (gt′, segs′)

uAcc_local(k, φ, ψ_l, ψ_r, seg)
  stack ← ∅; d ← −∞;
  for i ← seg.size − 1 to 0: begin
    if isLeaf(seg[i]) then seg′[i] ← seg[i]; stack ← seg′[i]; d ← d + 1; endif
    if isNode(seg[i]) then
      lv ← stack; rv ← stack;
      if d == 0      then stack ← ψ_l(lv, φ(seg[i]), rv); d ← 0;
      else if d == 1 then stack ← ψ_r(lv, φ(seg[i]), rv); d ← 0;
      else                seg′[i] ← k(lv, seg[i], rv); stack ← seg′[i]; d ← d − 1; endif
    endif
    if isCritical(seg[i]) then stack ← φ(seg[i]); d ← 0; endif
  end
  top ← stack; return(top, seg′);

uAcc_global(ψ_n, gt)
  stack ← ∅;
  for i ← gt.size − 1 to 0: begin
    if isLeaf(gt[i]) then gt′[i] ← gt[i]; endif
    if isNode(gt[i]) then lv ← stack; rv ← stack; gt′[i] ← ψ_n(lv, gt[i], rv); endif
    stack ← gt′[i];
  end
  return(gt′);

uAcc_update(k, seg, seg′, (lc, rc))
  stack ← ∅; stack ← rc; stack ← lc; d ← −∞;
  for i ← seg.size − 1 to 0: begin
    if isLeaf(seg[i]) then stack ← seg′[i]; d ← d + 1; endif
    if isNode(seg[i]) then
      lv ← stack; rv ← stack;
      if d == 0      then seg′[i] ← k(lv, seg[i], rv); stack ← seg′[i];
      else if d == 1 then seg′[i] ← k(lv, seg[i], rv); stack ← seg′[i]; d ← 0;
      else                stack ← seg′[i]; d ← d − 1; endif
    if isCritical(seg[i]) then
      lv ← stack; rv ← stack;
      seg′[i] ← k(lv, seg[i], rv); stack ← seg′[i]; d ← 0;
    endif
  end
  return(seg′);
```

Fig. 4.3. *Implementation of* uAcc *skeleton*

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

$$t_1(\textsc{uAcc_update}) = D_i \times t_1(k) \ .$$

In summary, using the functions defined so far, we can implement the uAcc skeleton. The cost of the uAcc skeleton is given as follows.

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

On the BSP model, we can implement the uAcc skeleton with three supersteps: the first one consists of Steps 1 and 2; the second one consists of Steps 3 and 4; the last one consists of Step 5. Thus, the BSP cost is given as follows.

$$\begin{aligned} & t_P^{\langle \text{BSP} \rangle}(\mathsf{uAcc}\ k) \\ &\quad = \max_p \sum_{pr(i)=p} \left(L_i \times \frac{t_1(k)}{2} + D_i \times (t_1(\phi) + \max(t_1(\psi_l), t_1(\psi_r))) \right) \\ &\qquad + M \times t_1(\psi_n)/2 + M \times (3|\alpha| + |\gamma|)/2 \times g + 3l \end{aligned}$$

4.4. Downwards Accumulate. Finally, we develop an implementation and the cost of the dAcc skeleton called with a pair of 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 input binary tree have type $BTree\langle \alpha, \beta \rangle$, the accumulative parameter c have type γ, and the intermediate values for auxiliary functions have type δ. The implementation of the dAcc skeleton is shown in Fig. 4.4.

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

In this step we apply ψ_u twice and either ϕ_l or ϕ_r for each of the ancestors of the m-critical nodes (D_i nodes). Omitting some small constants, the cost of the dAcc_path function is given as

$$t_1(\textsc{dAcc_path}) = D_i \times (\max(t_1(\phi_l), t_1(\phi_r)) + 2t_1(\psi_u)) \ .$$

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

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

The pair of local results from each internal segment is 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 dAcc_global with a forward traversal using a stack. Firstly, the initial value of accumulative parameter is pushed to the stack, and then the

accumulative parameter in the stack is updated with the pair of local results given in the previous step. The result of global accumulation is the accumulative parameter passed to the root node for each segment.

The dAcc_global function applies function ψ_d twice for each internal segment in the global structure. The computational cost of the dAcc_global function is

$$t_1(\text{dAcc_global}) = M \times t_1(\psi_d)\ .$$

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

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

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 dAcc_local function is just the same as the sequential version of the downwards accumulation on the serialized array if we assume the m-critical node as a leaf.

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

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

Summarizing the discussion so far, the cost of the dAcc skeleton is given as follows.

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

On the BSP model, we can implement the dAcc skeleton with three supersteps: the first one consists of Steps 1 and 2; the second one consists of Steps 3 and 4; the last one consists of Step 5. Thus, the BSP cost is given as follows.

$$\begin{aligned} &t_P^{\langle \text{BSP} \rangle}(\mathsf{dAcc}) \\ &\quad = \max_p \sum_{pr(i)=p} \left(L_i \times \frac{t_1(g_l) + t_1(g_r)}{2} + D_i \times (\max(t_1(\phi_l), t_1(\phi_r)) + 2t_1(\psi_u)) \right) \\ &\quad\quad + M \times t_1(\psi_d) + M \times (|\delta| + |\gamma|) \times g + 3l \end{aligned}$$

5. Optimal Division of Binary Trees Based on Cost Model. As we stated at the beginning of Sect. 3, locality and load balance are two important issues in developing efficient parallel programs in particular on distributed-memory parallel computers. When we divide and distribute a binary tree using the m-bridges, we enjoy good locality with large m while we enjoy good load balance with small m. Therefore, we need to find an appropriate value for m to achieve both good locality and good load balance.

First, we show relations among parameters of the cost model. From Lemma 3.5 and the representation of local segments in Fig. 3.2, we have

$$L_i \le m\ . \tag{5.1}$$

```
dAcc(g_l, g_r, c, (gt, segs)) where (g_l, g_r) = ⟨φ_l, φ_r, ψ_u, ψ_d⟩_d
  for i ← 0 to gt.size − 1: begin
    if pr(i) == p ∧ isNode(gt[i]) then
      gt[i] ← dAcc_path(φ_l, φ_r, ψ_u, segs[i]); endif
  end
  gather_to_root(gt)
  if isRoot(p) then gt′ ← dAcc_global(ψ_d, c, gt); endif
  distribute_from_root(gt′)
  for i ← 0 to gt.size − 1: begin
    if pr(i) == p then segs′[i] ← dAcc_local(g_l, g_r, gt′[i], segs[i]); endif
  end
  return (gt′, segs′)

dAcc_path(φ_l, φ_r, ψ_u, seg)
  d ← −∞;
  for i ← seg.size − 1 to 0: begin
    if isLeaf(seg[i]) then d ← d + 1; endif
    if isNode(seg[i]) then
      if d == 0 then
        toL ← ψ_u(φ_l(seg[i]), toL); toR ← ψ_u(φ_l(seg[i]), toR);
      else if d == 1 then
        toL ← ψ_u(φ_r(seg[i]), toL); toR ← ψ_u(φ_r(seg[i]), toR); d ← 0;
      else
        d ← d − 1;
      endif
    endif
    if isCritical(seg[i]) then toL ← φ_l(seg[i]); toR ← φ_r(seg[i]); d ← 0; endif
  end
  return (toL, toR);

dAcc_global(ψ_d, c, gt)
  stack ← ∅; stack ← c;
  for i ← 0 to gt.size − 1: begin
    if isLeaf(gt[i]) then gt′[i] ← stack; endif
    if isNode(gt[i]) then
      gt′[i] ← stack; (toL, toR) ← gt[i];
      stack ← ψ_d(gt′[i], toR); stack ← ψ_d(gt′[i], toL);
    endif
  end
  return gt′;

dAcc_local(g_l, g_r, c′, seg)
  stack ← ∅; stack ← c′;
  for i ← 0 to seg.size − 1: begin
    if isLeaf(seg[i]) then seg′[i] ← stack; endif
    if isNode(seg[i]) then
      seg′[i] ← stack; stack ← g_r(seg′[i], seg[i]); stack ← g_l(seg′[i], seg[i]); endif
    if isCritical(seg[i]) then seg′[i] ← stack; endif
  end
  return seg′;
```

Fig. 4.4. *Implementation of* dAcc *skeleton*

Since the height of a tree is at least a half of the number of nodes, we obtain

$$D_i \leq L_i/2 \leq m/2 \ . \tag{5.2}$$

From Lemmas 3.6 and 3.7, the number of local segments M is bound with the number N of nodes and the parameter m as follows.

$$\frac{1}{2}\left(\frac{N}{m} - 1\right) \leq M \leq \frac{2N}{m} - 1 \tag{5.3}$$

By inequality (5.2), the general form of the cost can be transformed into the following simpler form by considering the worst case.

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

We then bound the maximum number of nodes on a processor, $\max_p \sum_{pr(i)=p} L_i$. We distribute the local segment to processors so as to obtain good load balance, and one easy way to implement the load balancing is greedy distribution of the local segments from the largest one. By this greedy distribution, the difference between the maximum number of nodes $\max_p \sum_{pr(i)=p} L_i$ and the minimum number of nodes $\min_p \sum_{pr(i)=p} L_i$ is less than or equal to the maximum number of nodes in a segment. Since the maximum number of nodes in a local segment is m as stated in inequality (5.1) and the total number of nodes in the original binary tree is N, we can bound the maximum number of nodes distributed to a processor 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, we can bound the cost of the worst case.

$$\max_p \sum_{pr(i)=p} (L_i \times t_l + D_i \times t_d) + M \times t_m \leq \left(\frac{N}{P} + m\right) \times \left(t_l + \frac{t_d}{2}\right) + M \times t_m \tag{5.4}$$

Now we want to minimize the worst-case cost given in the right-hand side of inequality (5.4). By substituting the parameter M (inequality (5.3)), 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.

$$\sqrt{\frac{t_m}{2t_l + t_d}}\sqrt{N} \leq m \leq 2\sqrt{\frac{t_m}{2t_l + t_d}}\sqrt{N} \tag{5.5}$$

This range of the parameter m is much smaller than that used in the previous studies [8, 18, 30]. In Sect. 6, we will show experiment results that support the range.

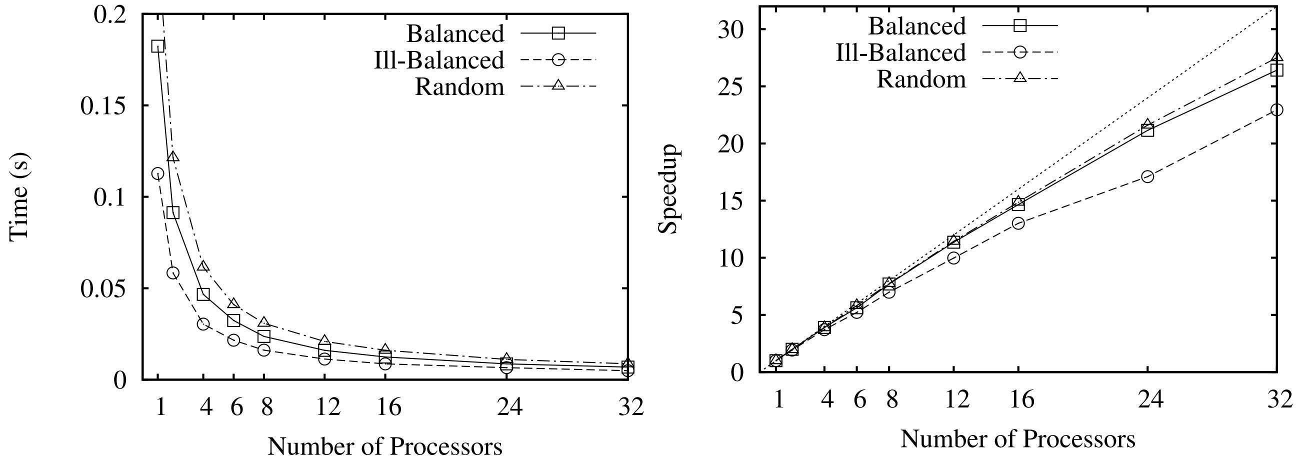

Fig. 6.1. *For the sum of node values, execution times and speedups against sequential program plotted to the number of processors. The parameter m for the division of trees is $m =$ 53,600.*

6. Experiment Results. To confirm the efficiency of the implementation algorithm for binary-tree skeletons, we implemented binary tree skeletons in C++ and MPI and made several experiments. We used our PC-cluster of uniform PCs with two Pentium 4 2.4-GHz CPUs (one CPU is used for each PC) 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 programs of the two examples in Sect. 2. The input trees are (1) a balanced tree, (2) a randomly generated tree and (3) a fully ill-balanced tree, each having 16,777,215 ($= 2^{24} - 1$) nodes.

Figures 6.1 and 6.2 show the general performance of 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, which is implemented on the array representing binary trees based on the same sequential algorithm. As seen in these plots, our implementation shows good scalability even against the efficient sequential programs. By the m-bridges, the balanced tree is divided into leaf segments of the same size and internal segments consisting only of one node. Therefore, the overhead caused by parallelism is very small for the balanced binary tree, and the implementation achieves almost linear speedups against the sequential program. For the random tree, the average depth of the m-critical nodes is so small that the implementation achieves good performance close to that for the balance tree. The fully ill-balanced tree, however, is divided into leaf segments consisting of one node and internal segments with their m-critical node at the depth $D_i = L_i/2$. From the cost model and its parameters, the skeletal parallel program has overheads caused by the factor of depth of the m-critical nodes. In fact, the experimental results show that the skeletal parallel program runs slower for the fully ill-balanced tree than for the other two inputs.

To analyze the cost model and the range of the parameter m more in detail, we made more experiments for the randomly generated tree by changing the value of the parameter m. We measured the parameters t_l, t_d, and t_m of the cost model with a small tree with 999,999 nodes, and estimated the value of them as $t_l = 0.057$ μs, $t_d = 0.03$ μs, and $t_m = 71$ μs. By substituting the parameters, we can expect good performance of the skeletal program under the range $90{,}000 < m < 180{,}000$. Figure 6.3 (left) plots the execution times to the number of processors for three values of the parameter m. As we can see from this figure, the implementation achieves good performance for a wide range of m. Figure 6.3 (right) plots the execution times to the parameter m. This figure shows that the performance gets worse if the parameter m is too small ($m <$ 5,000) or too large ($m >$ 200,000). For the parameter m in the range above the skeletal program achieves near the best performance, and we conclude that the cost model and the estimation of the parameter m is useful for efficient implementations.

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

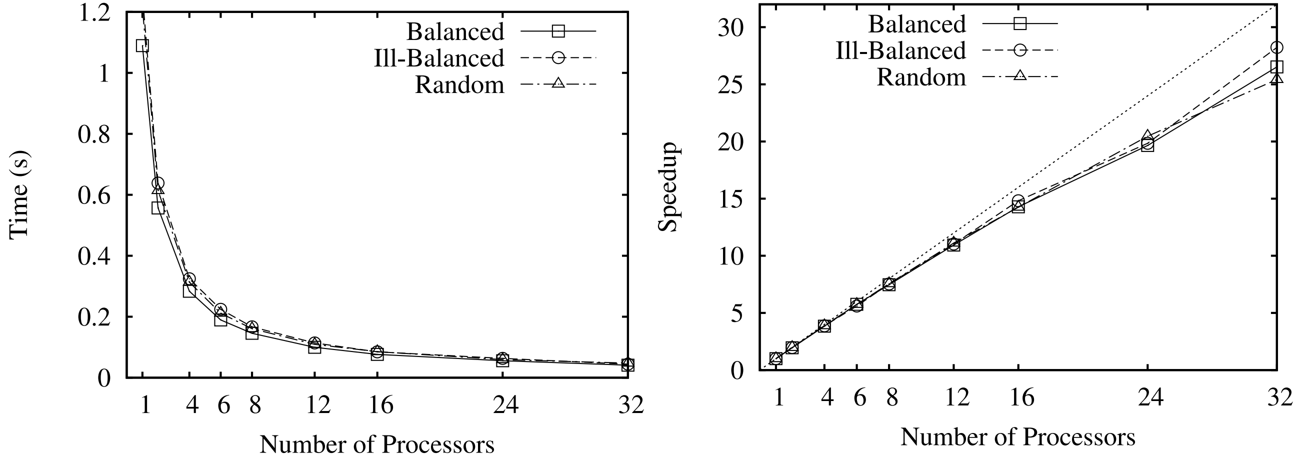

FIG. 6.2. *For the prefix numbering problem, execution times and speedups against sequential program plotted to the number of processors. The parameter m for the division of trees is $m = 53{,}600$.*

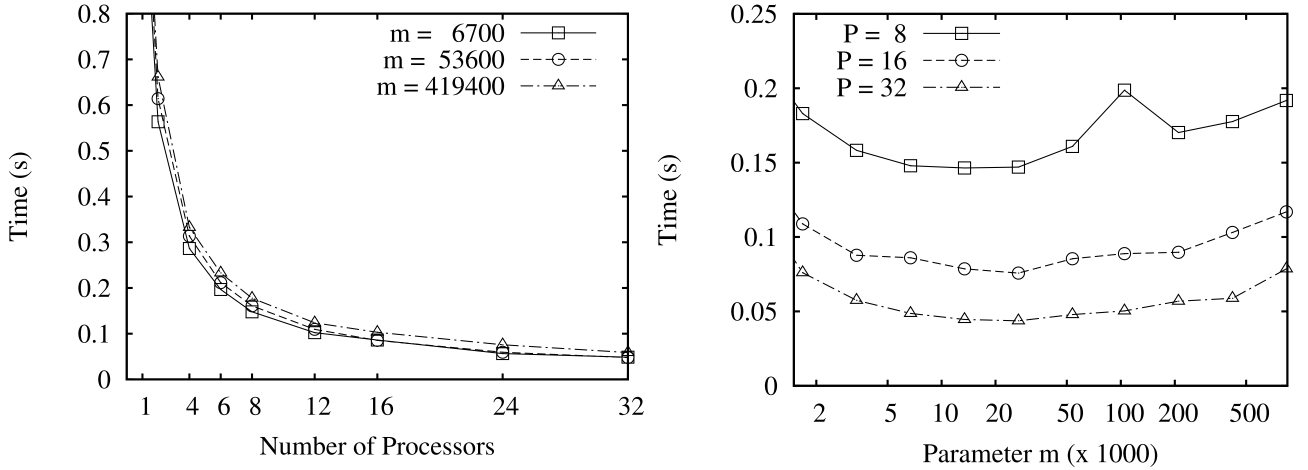

FIG. 6.3. *For the prefix numbering problem, execution times plotted to the number of processors and to the parameter m. The input trees are from the same randomly generated tree divided with different parameter m.*

to developing efficient implementations of tree contraction algorithms on various parallel models [1, 2, 3, 4, 5, 9, 13, 23, 24, 38]. Among them, Gibbons and Rytter developed a cost-optimal algorithm on CREW PRAM [9]; Abrahamson et al. developed a 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, 4, 9, 12, 17, 26, 27, 28, 29].

There have been several studies on the implementations of parallel tree skeletons [10, 11, 15, 18, 32, 33, 34]. Gibbons et al. [11, 33] have developed an implementation of tree skeletons based on tree contraction algorithms. Their algorithm can be used on many parallel computers, due to various implementation algorithms of tree contraction algorithms on various parallel computers. Skillicorn [34] and our previous paper [18] have discussed implementations of 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 tree skeletons as a parallel skeleton library. Our implementation of tree skeletons will be available as a part of SkeTo library [22]. Based on the implementation of the earlier work, Sato and Matsuzaki [31] improved its interface to support flexible manipulation of trees.

In terms of manipulations of general trees, which are formalized as parallel rose-tree skeletons [20], some of them are implemented efficiently in parallel [15, 32]. Sevilgen et al. [32] has shown an implementation algorithm for tree accumulations on general trees where rather strict conditions are requested for efficient implementation. Kakehi et al. [14] 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 is available as part of SkeTo library (`http://sketo.ipl-lab.org/`). One of our future work is to develop a profiling system to determine the parameter m more accurately.

REFERENCES

[1] K. R. Abrahamson, N. Dadoun, D. G. Kirkpatrick, and T. M. Przytycka, *A simple parallel tree contraction algorithm*, J. Algorithm., 10 (1989), pp. 287–302.

[2] D. A. Bader, S. Sreshta, and N. R. Weisse-Bernstein, *Evaluating arithmetic expressions using tree contraction: A fast and scalable parallel implementation for symmetric multiprocessors (SMPs) (extended abstract)*, in Proceedings of 9th International Conference on High Performance Computing (HiPC 2002), Springer, 2002, pp. 63–78.

[3] R. P. K. Banerjee, V. Goel, and A. Mukherjee, *Efficient parallel evaluation of CSG tree using fixed number of processors*, in ACM Symposium on Solid Modeling Foundations and CAD/CAM Applications, May 19–21, 1993, Montreal, Canada, 1993, pp. 137–146.

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

[5] F. K. H. A. Dehne, A. Ferreira, E. Cáceres, S. W. Song, and A. Roncato, *Efficient parallel graph algorithms for coarse-grained multicomputers and BSP*, Algorithmica, 33 (2002), pp. 183–200.

[6] H. Deldari, J. R. Davy, and P. M. Dew, *Parallel CSG, skeletons and performance modeling*, in Proceedings of the Second Annual CSI Computer Conference (CSICC'96), 1996, pp. 115–122.

[7] K. Diks and T. Hagerup, *More general parallel tree contraction: Register allocation and broadcasting in a tree*, Theor. Comput. Sci., 203 (1998), pp. 3–29.

[8] H. Gazit, G. L. Miller, and S.-H. Teng, *Optimal tree contraction in EREW model*, in Proceedings of the Princeton Workshop on Algorithms, Architectures, and Technical Issues for Models of Concurrent Computation, 1987, pp. 139–156.

[9] A. Gibbons and W. Rytter, *An optimal parallel algorithm for dynamic expression evaluation and its applications*, in Proceedings of the 6th Conference on Foundations of Software Technology and Theoretical Computer Science, Springer, 1986, pp. 453–469.

[10] J. Gibbons, *Computing downwards accumulations on trees quickly*, in Proceedings of the 16th Australian Computer Science Conference, 1993, pp. 685–691.

[11] J. Gibbons, W. Cai, and D. B. Skillicorn, *Efficient parallel algorithms for tree accumulations*, Sci. Comput. Program., 23 (1994), pp. 1–18.

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

[13] G. Jens, *Communication and memory optimized tree contraction and list ranking*, Tech. report, INRIA, Unité de recherche, Rhône-Alpes, Montbonnot-Saint-Martin, FRANCE, 2000.

[14] K. Kakehi, K. Matsuzaki, and K. Emoto, *Efficient parallel tree reductions on distributed memory environments*, in Proceedings of the 7th International Conference on Computational Science (ICCS 2007), Part II, Springer, 2007, pp. 601–608.

[15] K. Kakehi, K. Matsuzaki, K. Emoto, and Z. Hu, *An practicable framework for tree reductions under distributed memory environments*, Tech. Report METR 2006-64, Department of Mathematical Informatics, Graduate School of Information Science and Technology, the University of Tokyo, 2006.

[16] K. Matsuzaki, *Efficient implementation of tree accumulations on distributed-memory parallel computers*, in Proceedings of the 7th International Conference on Computational Science (ICCS 2007), Part II, Springer, 2007, pp. 609–616.

[17] K. Matsuzaki, Z. Hu, K. Kakehi, and M. Takeichi, *Systematic derivation of tree contraction algorithms*, Parallel Process. Lett., 15 (2005), pp. 321–336.

[18] K. Matsuzaki, Z. Hu, and M. Takeichi, *Implementation of parallel tree skeletons on distributed systems*, in Proceedings of The Third Asian Workshop on Programming Languages and Systems (APLAS'02), 2002, pp. 258–271.

[19] ———, *Parallelization with tree skeletons*, in Proceedings of the 9th International Euro-Par Conference (Euro-Par 2003), Springer, 2003, pp. 789–798.

[20] ———, *Parallel skeletons for manipulating general trees*, Parallel Comput., 32 (2006), pp. 590–603.

[21] ———, *Towards automatic parallelization of tree reductions in dynamic programming*, in Proceedings of the 18th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2006), ACM Press, 2006, pp. 39–48.

[22] K. Matsuzaki, H. Iwasaki, K. Emoto, and Z. 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, vol. 152 of ACM International Conference Proceeding Series, ACM Press, 2006.

[23] E. W. Mayr and R. Werchner, *Optimal routing of parentheses on the hypercube*, J. Parallel Dist. Com., 26 (1995), pp. 181–192.
[24] ———, *Optimal tree contraction and term matching on the hypercube and related networks*, Algorithmica, 18 (1997), pp. 445–460.
[25] G. L. Miller and J. H. Reif, *Parallel tree contraction and its application*, in Proceedings of 26th Annual Symposium on Foundations of Computer Science, IEEE Computer Society, 1985, pp. 478–489.
[26] ———, *Parallel tree contraction, part 2: Further applications*, SIAM J. Comput., 20 (1991), pp. 1128–1147.
[27] G. L. Miller and S.-H. Teng, *Dynamic parallel complexity of computational circuits*, in Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing, ACM Press, 1987, pp. 254–263.
[28] ———, *Tree-based parallel algorithm design*, Algorithmica, 19 (1997), pp. 369–389.
[29] ———, *The dynamic parallel complexity of computational circuits*, SIAM J. Comput., 28 (1999), pp. 1664–1688.
[30] J. H. Reif, ed., *Synthesis of Parallel Algorithms*, Morgan Kaufmann Publishers, February 1993.
[31] S. Sato and K. Matsuzaki, *A generic implementation of tree skeletons*, International Journal of Parallel Programming, 44 (2016), pp. 686–707.
[32] F. E. Sevilgen, S. Aluru, and N. Futamura, *Parallel algorithms for tree accumulations*, J. Parallel Dist. Com., 65 (2005), pp. 85–93.
[33] D. B. Skillicorn, *Foundations of Parallel Programming*, vol. 6 of Cambridge International Series on Parallel Computation, Cambridge University Press, 1994.
[34] ———, *Parallel implementation of tree skeletons*, J. Parallel Dist. Com., 39 (1996), pp. 115–125.
[35] ———, *A parallel tree difference algorithm*, Inform. Process. Lett., 60 (1996), pp. 231–235.
[36] ———, *Structured parallel computation in structured documents*, J. Univars. Comput. Sci., 3 (1997), pp. 42–68.
[37] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl, *Questions and Answers about BSP*, Scientific Programming, 6 (1997), pp. 249–274.
[38] U. Vishkin, *A no-busy-wait balanced tree parallel algorithmic paradigm*, in Proceedings of the 12th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 2000), ACM Press, 2000, pp. 147–155.

Edited by: Frédéric Loulergue
Received: September 17, 2016
Accepted: January 17, 2017