Efficient Parallel Tree Reductions on Distributed Memory Environments

Kazuhiko Kakehi[1], Kiminori Matsuzaki[2], and Kento Emoto[3]

[1] Division of University Corporate Relations
[2] Department of Mathematical Informatics
[3] Department of Creative Informatics
University of Tokyo
{kaz,kmatsu,emoto}@ipl.t.u-tokyo.ac.jp

Abstract. A new approach for fast parallel reductions on trees over distributed memory environments is proposed. By employing serialized trees as the data representation, our algorithm has a communication-efficient BSP implementation regardless of the shapes of inputs. The prototype implementation supports its real efficacy.

Keywords: Tree reduction, parentheses matching, BSP, parallel algorithm.

1 Introduction

Research and development of parallelization have been intensively done toward matrices or one dimensional arrays. Looking at recent trends in applications, another data structure has also been calling for efficient parallel treatments: the tree structure. Emergence of XML as a universal data format, which takes the form of a tree, has magnified the impact of parallel and distributed mechanisms toward trees in order to reduce computation time and mitigate limitation of memory. However, parallel tree computation over distributed memory environments is not so straightforward as it looks.

Consider, as a simple and our running example, a computation *maxPath* to find the maximum of the values each of which is a sum of values in the nodes from the root to each leaf. When it is applied to the tree in Fig. 1, the result should be 12 contributed by the path of values 3, -5, 6 and 8 from the root. Parallelization of such a simple computation under distributed memory environments requires consideration from two aspects. The first is the underlying data representations, including its distribution among processors to guarantee performance toward trees of arbitrary shapes. The second is derivation of the parallel algorithm. As associativity often helps parallelization, we need to exploit the similar property under trees which suitably work for the data representations.

This paper gives a clear solution for parallel tree reductions with its start point to use *serialized forms of trees*. Their Notable examples are the serialized (streamed) representations of XML or parenthesized numeric expressions which are obtained by tree traversals. The first problem mentioned above is naturally resolved by this choice, since distribution of serialized data among processors is much simpler than that of trees. As for the second point, we present an efficient parallel algorithm for tree reductions satisfying *extended distributivity*. As instances of serialized trees, parallelization of *parentheses matching problems*, which figures out correspondence between brackets, have

Y. Shi et al. (Eds.): ICCS 2007, Part II, LNCS 4488, pp. 601–608, 2007.

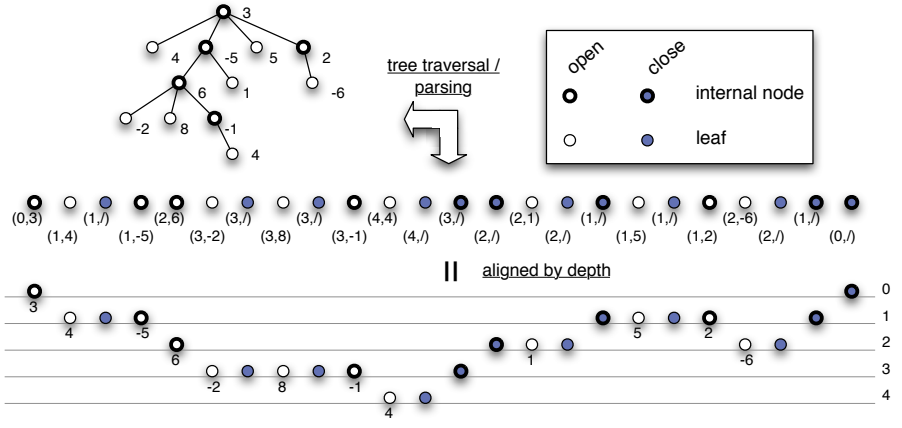


Fig. 1. A rose tree (upper left), its serialized representation as a sequence of pairs (middle) and another representation according to the depth (lower)

plenty of work ([1, 2, 3, 4, 5] to mention a few); our algorithm, with good resemblance to the one under BSP [6], has also a BSP implementation with three supersteps.

This paper is organized as follows. After this Introduction, Sect. 2 observes our tree representations and tree reductions. Section 3 develops the algorithm. Our algorithm consists of three phases, where the first two perform computation along with parentheses matching, and the last reduces a tree of size less than twice of the number of processors. Section 4 supports our claims by some experiments. We conclude this paper in Sect. 5 and mentioning future directions. We refer the interested reader to our technical report [7] for the omitted details due to the space limitation of this paper.

2 Preliminaries

This section observes the underlying frameworks for our parallelization: tree structures, their serialized form, tree homomorphism, and extended distributivity.

2.1 Trees and Their Serialized Representation

We treat trees with unbound degree (trees whose nodes can have an arbitrary number of subtrees); Fig. 1 shows an example. As was explained in Introduction, our internal representation is to keep tree-structured data in a serialized manner. The sequence of the middle in Fig. 1 is our internal representation of the example tree. Like XML serialization or parentheses expressions, it is a combination of a preorder (for producing the open elements) and a postorder traversal (for producing the close elements afterwards). We assume *well-formedness*, with which sequences are guaranteed to be parsed back into trees, and without loss of information we simplify close elements to be "/". For the convenience of later discussion, we assign the information of the depth in the tree to each node. The figure also depicts their presentation according to the depth.

2.2 Tree Homomorphism and Extended Distributivity

We use the framework called *tree homomorphism* [8], which specifies recursive tree reductions h using h', $\oplus$, and associative $\otimes$ with its unit $\iota_\otimes$:

$$\begin{aligned} h(\ \mathit{Node}\ a\ [t_1, \ldots, t_n]\ ) &= a \oplus (\ h(t_1) \otimes \cdots \otimes h(t_n)\ )\ , \\ h(\ \mathit{Leaf}\ a\ ) &= h'(a)\ . \end{aligned}$$

The computation *maxPath* mentioned in Introduction is also a tree homomorphism:

$$\begin{aligned} \mathit{maxPath}(\ \mathit{Node}\ a\ [t_1, \ldots, t_n]\ ) &= a + (\ \mathit{maxPath}(t_1) \uparrow \cdots \uparrow \mathit{maxPath}(t_n)\ )\ , \\ \mathit{maxPath}(\ \mathit{Leaf}\ a\ ) &= id(a) = a\ . \end{aligned}$$

Here, id is the identify function, and $\uparrow$ returns the bigger of two numbers whose unit is $-\infty$. When it is applied to the tree in Fig. 1, the result should be $12 = 3 + (-5) + 6 + 8$.

We can apply parallelization of tree homomorphism over serialized trees based on list homomorphism [3]. This naive approach, however, suffers from the factor of tree depths. We need to review an additional property called *extended distributivity* [9]. This property is explained, by introducing a new operator $\ominus$ defined as $(a, b, c) \ominus e = a \oplus (b \otimes e \otimes c)$, as follows: for any *triple*s (a_u, b_u, c_u), (a_l, b_l, c_l), and any expression e, there exists a *triple* (a, b, c) which satisfies

$$(a_u, b_u, c_u) \ominus (\ (a_l, b_l, c_l) \ominus e\ ) = (a, b, c) \ominus e\ .$$

Efficient parallel reduction requires these computations as well as $\otimes$ and $\oplus$ to be done in constant time. Our running example satisfies these properties as the following calculation shows.

$$\begin{aligned} &(a_u, b_u, c_u) \ominus (\ (a_l, b_l, c_l) \ominus e\ ) \\ &= a_u + (b_u \uparrow (a_l + (b_l \uparrow e \uparrow c_l)) \uparrow c_u) \\ &= (a_u + a_l) + ((-a_l + b_u \uparrow b_l) \uparrow e \uparrow (c_l \uparrow -a_l + c_u)) \\ &= (\ (a_u + a_l),\ (-a_l + b_u \uparrow b_l),\ (c_l \uparrow -a_l + c_u)\ ) \ominus e \end{aligned}$$

For some other examples under these formalizations, see our previous work [9].

3 Parallel Computation over the Serialized Trees

This section develops a parallel algorithm for tree homomorphism which satisfies extended distributivity. We explain our algorithm in three phases: (1) the first phase applies tree homomorphism toward segments (consecutive subsequences) in each processor as much as possible; (2) after communications among processors the second phase performs further reduction using extended distributivity, producing a binary tree as a result whose internal nodes are specified as *triple*s, and size is less than twice of the number of processors; finally (3) the third phase reduces the binary tree into a single value.

We use N and P to the number of tree nodes and available processors, respectively. During the explanation we assume $\mathrm{P} = 4$. The algorithm resembles the analysis of parentheses matching problems under BSP [5].

First phase. Each processor applies tree homomorphism to its given segment of size $2\mathrm{N}/\mathrm{P}$. The process is summarized as Routine 1. This process leaves fragments of results,

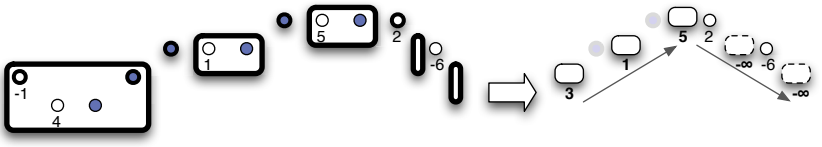


Fig. 2. An illustrating example of the first phase—applying tree homomorphism *maxPath* to a segment of Fig. 1 (lined and dashed ovals indicate values obtained by reduction and $-\infty$ (the unit of $\uparrow$), respectively)

in arrays as_p, bs_p, cs_p and an integer d_p for each processor number p. The array as_p is to keep the open elements without their corresponding close element. Each element of as_p can have subtrees before the next one in as_p, and their reduced values are kept in bs_p. Similar treatments are done to unmatched close elements, leaving values in cs_p (we remove unmatched close elements thanks to the absence of values). The integer d_p denotes the shallowest depth in processor p. While both of elements in as_p and bs_p are listed in a descending manner, those of cs_p are in ascending manner; the initial elements of as_p and bs_p and the last one of cs_p are at height d_p (except for cs_0 and $cs_{\mathrm{P}-1}$ whose last element at depth 0 is always $\iota_\otimes$ and therefore is set to be eliminated).

Routine 1. Prepare arrays as, bs, cs (behaving as stacks growing from left to right), and an integer variable d. A temporary t is used. Sequence $(d_0, a_0), \ldots, (d_{n-1}, a_{n-1})$ $(n \geq 1)$ is given.

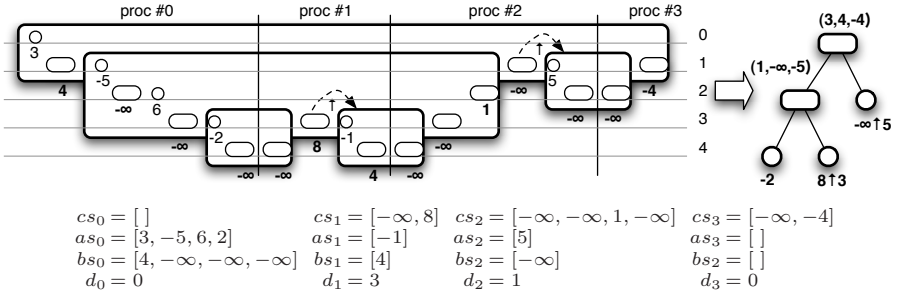
- Set $d \leftarrow d_0$. If a_0 is "/" then $cs \leftarrow [\iota_\otimes, \iota_\otimes]$, $as \leftarrow [\,]$, $bs \leftarrow [\,]$; else $cs \leftarrow [\iota_\otimes]$, $as \leftarrow [a_0]$, $bs \leftarrow [\iota_\otimes]$.
- For each i in $\{1, \ldots, n-1\}$
  - if a_i is not "/" (namely a value), then push a_i to as and $\iota_\otimes$ to bs;
    else • if as is empty, then push $\iota_\otimes$ to cs, and set $d \leftarrow d_i$;
      else • pop a' from as and b' from bs.
        - if $b' = \iota_\otimes$ (implying a' is a leaf) then $t \leftarrow h'(a')$; else $t \leftarrow a' \otimes b'$;
        - if bs is not empty, then pop b'' from bs and push $b'' \otimes t$ to bs;
          else pop c'' from cs and push $c'' \otimes t$ to cs.
- If $\mathrm{P} \neq 1$ then remove $\iota_\otimes$ at the bottom of cs_0 and $cs_{\mathrm{P}-1}$.

In Fig. 2 we show a case of the illustrating segment from the 10th element $(3, -1)$ to the 21st $(2, -6)$ of the sequence in Fig. 1. We have, as depicted:

$$\begin{aligned} cs &= [-1 + id(4),\ id(1),\ id(5)] & as &= [2, 6], & d &= 1. \\ &= [3, 1, 5], & bs &= [-\infty, -\infty], \end{aligned}$$

Please note that we regard absence of subtrees as an empty forest to which the tree homomorphism returns $-\infty$, the unit of $\uparrow$ (at depth 2 and 3 kept in bs). When we distribute the whole sequence in Fig. 1 evenly among four processors (6 elements for each), the results by this phase is shown in Fig. 3, left.

A linear routine produces as, bs, cs and d. The worst case in terms of the size of the results occurs when an sequence of only open elements is given, resulting in two arrays as and bs of the length $2\mathrm{N}/\mathrm{P}$ for each.



$cs_0 = [\,]$ $as_0 = [3, -5, 6, 2]$ $bs_0 = [4, -\infty, -\infty, -\infty]$ $d_0 = 0$

$cs_1 = [-\infty, 8]$ $as_1 = [-1]$ $bs_1 = [4]$ $d_1 = 3$

$cs_2 = [-\infty, -\infty, 1, -\infty]$ $as_2 = [5]$ $bs_2 = [-\infty]$ $d_2 = 1$

$cs_3 = [-\infty, -4]$ $as_3 = [\,]$ $bs_2 = [\,]$ $d_3 = 0$

Fig. 3. *Triple*s between two processors (left) and the resulting tree (right) after the second phase

Second phase. The second phase matches data fragments kept in each processor into *triple*s using communication between processors. Later, we reduce consecutive occurrences of *triple*s into a value, or into one *triple* by extended distributivity.

When we look carefully at Fig. 3, we notice that 3 in as_0 at depth 0 now has five parts at depth 1 as its children: the value 4 in bs_0, a subtree spanning from processors 0 to 2 whose root is -5 in as_0, the value $-\infty$ in cs_2, a subtree from processor 2 to 3 whose root is 5 in as_2, and the value -4 in cs_3. As these subtrees need reducing separately, we focus on the leftmost and the rightmost values in bs_0 and cs_3 (we leave the value $-\infty$ in cs_2 for the time being). We notice that the *group* of the value 3 in as_0 with these two values in processors 0 and 3 forms a *triple* $(3, 4, -4)$.

Similarly, two elements in as_0 at depth 1 and 2, with two elements each in bs_0 and cs_2 at depth 2 and 3, respectively, form two *triple*s $(-5, -\infty, 1)$ and $(6, -\infty, -\infty)$. The former *triple* indicates a tree that awaits the result of one subtree specified by the latter. This situation is what extended distributivity takes care of: we can merge two *triple*s (a sequence of *triple*s in general) into one: $(-5, -\infty, 1) \oplus (\ (6, -\infty, -\infty) \oplus e\ ) = (\ (-5 + 6),\ (-6 - \infty \uparrow -\infty),\ (-\infty \uparrow -6 + 1)\ ) \oplus e = (1, -\infty, -5) \oplus e$ for any e. In this way, the *group* of data fragments in two processors turn into one *triple*.

Such *group*s from two adjacent processors are reduced into a single value without any missing subtrees in between: instead of treating using extended distributivity, the values -2 in as_0 and -1 in as_1 at depth 3, and 5 in as_2 at depth 2 with their corresponding values in bs_i and cs_{i+1} $(i = 0, 1, 2)$ turn into values $id(-2) = -2$, $-1 + (4 \uparrow -\infty) = 3$, $id(5) = 5$, respectively.

We state the following lemma to tell the number of resulting *groups* in total.

Lemma 1. *The second phase produces* group*s of the number at most* $2P - 3$.

The proof sketch is as follows. Let R_p be the the number of *group*s among p processors. As Fig. 3 indicates, $R_2 = 1$, and for $p > 2$ we derive $R_p \leq 1 + R_j + R_{p-j+1}$ with $1 < j < p$; hence we have $R_p \leq 2p - 3$. This lemma guarantees that, after transactions of data fragments among processors, one processor needs to take care of at most two such *group*s. The computed value at the shallowest depth d_p in each processor are associated to their right for later computation by $\uparrow$ (8 in cs_1, $-\infty$ in cs_2; see Fig. 3).

The following Routine 2 figures out *group*s among processors. $M^{p_l \leftrightarrow p_r}_{[d_u, d_l]}$ denotes a *group* between processors p_l and p_r whose data fragments span from the depth d_u until d_l (∞ in d_l indicates "everything starting from d_u"); $M^{\overleftrightarrow{\ }}_{[d_u, d_l]}$ is inserted as a dummy

group in case the same d appear among more than two consecutive processors, and assumed to be reduced into $\iota_\ominus$, a virtual left unit of $\ominus$ (namely $\iota_\ominus \ominus e = e$).

Routine 2. A stack is used whose top is referred as (p_s, d_s). Sequence (p, d_p) is given in the ascending order of p.

- Push the first pair $(0, 0)$ on a stack
- For each i in $\{1, \ldots, \mathrm{P} - 1\}$
  - prepare a variable $d \leftarrow \infty$.
  - while $d_i < d_s$, produce $M^{p_s \leftrightarrow i}_{[d_s, d]}$, set $d \leftarrow d_s$ and pop from the stack;
  - if $d_i = d_s$, then produce $M^{p_s \leftrightarrow i}_{[d_s, d]}$ and $M_{[\overleftrightarrow{d_i}, \overleftrightarrow{d_s}]}$, and pop from the stack.
    else produce $M^{p_s \leftrightarrow i}_{[d_i, d]}$.
  - push (i, d_i).
- Finally eliminate the last mating pair (that is $M_{[\overleftrightarrow{0}, \overleftrightarrow{0}]}$).

We summarize this phase. This phase consists of two steps. In the first all processors figure out the *group*s by Routine 2 and allocation of *group*s by any deterministic rule through sharing their depth information d_i. They require $\mathrm{O}(\mathrm{P})$ communication and computation costs in total. The second step is to apply further computation toward each *group*. In the worst case it is possible that a processor sends out its whole data fragments in it to all other $\mathrm{P} - 1$ processors, and receives two *group*s and evaluates each into one *triple* or value. Since the size of fragments in one processor and that of each *group* are at most $\mathrm{O}(\mathrm{N}/\mathrm{P})$, the cost for data transaction and computation is bound by $\mathrm{O}(\mathrm{N}/\mathrm{P})$.

Third phase. This last phase compiles obtained *triple*s or values and reduce them into a single value. As Fig. 3 shows, the obtained *triple*s and values in the previous phase form a binary tree of size $2\mathrm{P} - 3$ (including dummies by $M_{[\overleftrightarrow{0}, \overleftrightarrow{0}]}$). We collect *triple*s or values in one processor, and apply tree reduction in $\mathrm{O}(\mathrm{P})$ time.

In summary, we have three communication rounds (two in the second phase, one in the third phase). We conclude this section by stating the following theorem.

Theorem 1. *Tree homomorphism with extended distributivity has a BSP implementation with three supersteps of at most* $\mathrm{O}(\mathrm{P} + \mathrm{N}/\mathrm{P})$ *communication cost for each.*

4 Experiments and Discussion

We performed experiments using our prototype implementation using C++ and MPI. Specifications of our PC cluster are shown in Table 1, left. We simulated a simple query on rose trees where local computation of $\otimes$ and $\oplus$ were matrix multiplication-like operations over matrices of the size 10×10. We prepared randomly generated trees of three types, namely (RS) shallow, (RD) deep, and (M) a monadic tree (a tree-view of a list). The tree size was constrained by the memory size a machine has. The height of RS came from observations on XML documents [10]. We executed the program over each type using 2^i processors ($i = 0, \ldots, 6$).

Table 1 summarizes the results. First, it is natural that no difference existed in terms of costs of initial data distribution. As their plots in the left of Fig. 4 indicates, our algorithm exhibited good scalability for both of (RS) and (RD). Results of (M) fell

Table 1. Specification (left) and execution times (right) of our experiments

machine	2.8 GHz CPU, 2GB memory
network	Gigabit Ethernet
software	Linux 2.4.21, gcc 2.96, mpich 1.2.7

data: randomly generated trees of size 2,000,000
(RS) with maximum height 7
(RD) with maximum height of 5,000
(M) a monadic tree

		Deep (RD)		Shallow (RS)		Monadic (M)	
P	dist.	comp.	total	comp.	total	comp.	total
1	—	23.5	23.5	22.8	22.8	N.A.	N.A.
2	0.48	12.3	12.7	12.6	13.0	N.A.	N.A.
4	0.56	6.13	6.70	5.81	6.36	60.1	60.7
8	0.62	3.18	3.80	3.79	4.41	20.9	21.5
16	0.70	1.81	2.51	1.96	2.65	14.8	15.5
32	0.77	1.11	1.88	1.04	1.81	8.36	9.14
64	0.83	0.68	1.52	0.57	1.40	4.47	5.30

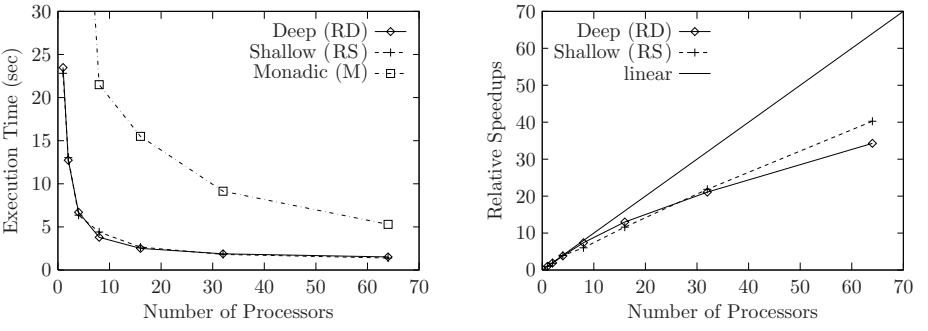


Fig. 4. Plots of Table 1 by total execution time (left) and by speedups of computation time (right)

behind the other two: it ran out of memory until 2 processors (the result by 4 processors seems also affected), and its execution was around seven times slower than other cases. The reason is that there are no reducible subtrees in the first phase, which incurs heavy data transactions and costly computation by extended distributivity. It should be noted that the similar BSP algorithm for *all nearest smaller values problem*, generalization of parentheses matching, toward average cases is shown not to suffer from heavy transactions [5]. While we need to develop a similar theoretical proof to our algorithm, experiments on random trees (RS and RD) suggest that our algorithm works efficiently toward average cases.

We make a brief comparison with existing approaches of parallel tree computations. One common approach under distributed memory environments has their basis in list ranking, and their parallel tree contractions require expensive costs of $O(N/P \log P)$ [11, 12, 13]. Using a technique called m-bridge [14] for initial data distribution, our existing library implements parallel tree contractions in $O(N/P + P)$ over trees kept as linked structures [15, 16, 9]. In comparison with them, the approach in this paper has two notable advantages. First, it can be coded as simple for-loops over one-dimensional arrays and benefits compiler optimizations well. We also observed that high memory locality by serialized forms brought considerable performance improvements, especially when the required computation was memory-intensive where cache effects become important. Second, the data distribution process in this paper is really small compared to the cost of m-bridge, in which traversal over linked structure is involved.

5 Concluding Remarks

In this paper we have developed a new approach to reduce a tree in parallel. The algorithm has been shown to run scalably as well as fast in theory and practice, by exploiting the serialized trees and a property called extended distributivity.

At the moment we have only analyzed cost for the worst case. It is our interesting task to analyze average cases, or go a step further to the theory of heterogeneous cases.

Acknowledgment. This research was partially supported by the Ministry of Education, Culture, Sports, Science and Technology, Grant-in-Aid for Young Scientists (B), 17700026, 2005–2007.

References

1. Berkman, O., Schieber, B., Vishkin, U.: Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. Journal of Algorithms **14** (1993)
2. Prasad, S., Das, S., Chen, C.: Efficient EREW PRAM algorithms for parentheses-matching. IEEE Transactions on Parallel and Distributed Systems **5**(9) (1994)
3. Cole, M.: Parallel programming with list homomorphisms. Parallel Processing Letters **5** (1995)
4. Kravets, D., Plaxton, C.: All nearest smaller values on the hypercube. IEEE Transactions on Parallel and Distributed Systems **7**(5) (1996)
5. He, X., Huang, C.: Communication efficient BSP algorithm for all nearest smaller values problem. Journal of Parallel and Distributed Computing **16** (2001)
6. Valiant, L.: A bridging model for parallel computation. Communication of the ACM **33**(8) (1990)
7. Kakehi, K., Matsuzaki, K., Emoto, K., Hu, Z.: A practicable framework for tree reduction under distributed memory environments. Technical Report METR 2006-64, Department of Mathematical Informatics, University of Tokyo (2006)
8. Skillicorn, D.B.: Parallel implementation of tree skeletons. Journal of Parallel and Distributed Computing **39**(2) (1996)
9. Matsuzaki, K., Hu, Z., Kakehi, K., Takeichi, M.: Systematic derivation of tree contraction algorithms. Parallel Processing Letters **15**(3) (2005) (Original version appeared in *Proc. 4th International Workshop on Constructive Methodology of Parallel Programming*, 2004.).
10. Mignet, L., Barbosa, D., Veltri, P.: The XML web: a fist study. In Proceedings of the Twelfth International World Wide Web Conference, ACM Press (2003)
11. Mayr, E.W., Werchner, R.: Optimal routing of parentheses on the hypercube. Journal of Parallel and Distributed Computing **26**(2) (1995)
12. Mayr, E.W., Werchner, R.: Optimal tree contraction and term matching on the hypercube and related networks. Algorithmica **18**(3) (1997)
13. Dehne, F., Ferreira, A., Cáceres, E., Song, S., Roncato, A.: Efficient parallel graph algorithms for coarse-grained multicomputers and BSP. Algorithmica **33**(2) (2002)
14. Reid-Miller, M., Miller, G.L., Modugno, F.: List ranking and parallel tree contraction. In Reif, J.H., ed.: Synthesis of Parallel Algorithms. Morgan Kaufmann (1993)
15. SkeTo Project Home Page. `http://www.ipl.t.u-tokyo.ac.jp/sketo/`
16. Matsuzaki, K., Emoto, K., Iwasaki, H., Hu, Z.: A library of constructive skeletons for sequential style of parallel programming (invited paper). In: Proceedings of the First International Conference on Scalable Information Systems, IEEE Press (2006)