IPSJ Transactions on Programming Vol.7 No.3 40–50 (July 2014)

Regular Paper

Dividing Huge XML Trees Using the *m*-bridge Technique over One-to-one Corresponding Binary Trees

Takayuki Kawamura[1] Kiminori Matsuzaki[1,a)]

Received: November 15, 2013, Accepted: March 19, 2014

Abstract: Tree data such as XML trees have recently been getting larger and larger. Parallel and distributed processing is a promising way of dealing with big data, but we need to divide the data in the first step. Since computation over trees often requires relationships between parents and children and/or among siblings, we should pay attention to such relationships. There is a technique called the "*m*-bridge" for dividing trees. We can easily compute *m*-bridges for trees of any shape. However, division with the *m*-bridge technique is sometimes unsatisfactory for shallow XML trees. We propose a method of tree division for XML trees in this study, in which we apply the *m*-bridge technique to a one-to-one corresponding binary tree. We implement the tree division algorithm using the Simple API for XML (SAX) Parser. An important feature of our algorithm is that we transform and divide XML trees in the order that the SAX parser reads the trees. We carried out experiments and discuss the properties of the tree division algorithm we propose. In addition, we discuss how we can use the divided trees with query examples.

Keywords: XML, distributed computing, data division, binary-tree representation, SAX

1. Introduction

A large amounts of data have recently been widely used, which calls for methods of processing such huge amounts of data. Tree data (or semi-structured data) like XML documents are also becoming larger [4].

It is often the case that a single computer has insufficient memory or storage to process such large amounts of data. Therefore, it is important to execute parallel or distributed computing with multiple computers. The first step in parallel or distributed computing is data division.

Many applications, which involve tree structures like XML, utilize the relations between parents and children. This means that tree division should focus on the relations between nodes. There is a tree division algorithm that holds some properties on the relations between parents and children, i.e., a tree division using *m*-bridges [14], [27]. We can divide a tree of any shape through simple computations with this tree division with *m*-bridges. A problem with tree division with *m*-bridges is that we sometimes have segments that are too small after shallow XML documents are divided.

To resolve this problem, we propose a tree-division algorithm in this study, by applying the *m*-bridge technique to a binary tree that has one-to-one correspondence to the XML tree. We can appropriately divide a tree even if the tree is very wide and shallow by computing *m*-bridges over binary-tree representations. We also implemented an existing tree division with *m*-bridges and the proposed division with *m*-bridges over the binary-tree representations by using the Simple API for XML (SAX) library [5]. We then conducted experiments with these programs and investigated the properties of tree division based on the binary-tree representations.

The three main contributions of the paper are summarized below.

- Based on the existing tree-division method with *m*-bridges (Section 2), we propose a new tree-division method with *m*-bridges over binary-tree representations (Section 3). We can apply the proposed tree-division method to XML trees of any shape. The number of parts divided by the proposed method is much smaller than that by the existing method.
- We implemented each of the existing tree divisions with *m*-bridges and the proposed tree division with *m*-bridges over the binary-tree representations as a one-pass algorithm using the SAX library (Section 4). The results from experiments demonstrated that our implementation with *m*-bridges over binary-tree representations could divide a tree sufficiently fast (Section 5).
- We also discuss how we can use the data divided by the proposed method in distributed computing (Section 6).

Distributed Processing Model That We Assume in This Study

The final goal of our research was to develop a tree-processing framework that could be used in cloud-computing environments. Except for cases where we have data in the cloud environment we use, we are generally required to send data to the cloud environment through networks. We implemented *m*-bridge-based tree-division with a one-pass algorithm using SAX in this study, so that we could divide the tree at the gateway of the cloud environment in a streaming-processing manner [*1].

We assumed MapReduce/Hadoop or related models as the dis-

[1] Kochi University of Technology, Kami, Kochi 782–8502, Japan
[a)] matsuzaki.kiminori@kochi-tech.ac.jp

[*1] If the algorithm consists of with more than one pass, then we need to store the data somewhere. This becomes a problem when we cannot place the (whole) data on the gateway.

tributed processing model. Data are split into blocks of constant size (64 MB in default) and managed by blocks in the Hadoop distributed file system (HDFS). We stored parts of data in different files in this study. Parts of data in HDFS are copied to some other nodes in the background, but we did not take into account the cost of this replication. Computation in the MapReduce programming model is performed in two phases: a function is applied independently to each part of the data (the map phase) followed by a process that merges the intermediate results (the reduce phase). The number of parts after division (number of divisions) strongly affects the performance of parallel computing: too few divisions may reduce the degree of parallelism; too many divisions may increase the overheads of calling the function in the map phase and the cost of the merge process in the reduce phase. The authors demonstrated in a previous study [17] that we could compute MapReduce faster on a divided binary tree transformed from a wide tree than on the original tree.

We assumed in the target computations of this study that we could merge the partial results locally among siblings. Note that such computations include the general tree reduction given in Ref. [23]. The example discussed in Section 6, i.e., an XPath query, also has this property.

2. Tree Division with *m*-bridges

First, we will introduce the existing method of tree division with *m*-bridges and its properties [14], [27]. In the rest of the paper, we denote the size of the subtree rooted at *x* as *x.size*. Since we dealt with XML in this study, the size of the subtree is the number of characters in the text representation. In terms of the up-down relation over trees, we call the side near the root *up*, and the side near the leaves *down*.

Definition 1 (*m*-critical node) Let N be the size of a tree, and m be an integer satisfying $1 < m \leq N$. Node x of the tree is called *m*-critical, if it satisfies the following conditions:

- x is an internal node and
- for each child c of x, inequality

$$\left\lceil \frac{x.size}{m} \right\rceil > \left\lceil \frac{c.size}{m} \right\rceil$$

holds. □

We can obtain *m*-bridges by dividing the tree beneath the *m*-critical nodes [*2].

Definition 2 (*m*-bridge) Let N be the size of a tree, and m be an integer satisfying $1 < m \leq N$. An *m*-bridge is a maximal segment that includes an *m*-critical node at the bottom (it becomes a leaf in the segment). □

Definition 3 (Global tree) When we divide a tree with *m*-bridges, we can obtain a tree by reducing each segment into a single node. We call the tree a *global tree*. □

As an example, we explain how we can divide an XML document

<a><a></a><b><c><a></a><b></b>text1</c>
<b><a></a></b></b><c></c><b><a></a><b>
text2</b></b><d><c></c></d><b a="1"></b></a>

with parameter $m = 35$. The tree is shown in **Fig. 1** where each number on the upper right of the node denotes the size of the subtrees rooted at the node, and where the size is equal to the length from the opening tag to the closing tag. For example, the size of the subtree in Fig. 1 rooted at node <c> at the leftmost in the third row is the length of <c><a></a><b></b>text1</c>, i.e., 26. The *m*-critical nodes in the tree in Fig. 1 are the two denoted by doubled circles. For example, at node <b> in the second from the left in the second row, two inequalities $\lceil 47/35 \rceil > \lceil 26/35 \rceil$ and $\lceil 47/35 \rceil > \lceil 14/35 \rceil$ hold. We divide the tree in tree division with *m*-bridges immediately beneath *m*-critical nodes (**Fig. 2**). The tree in the example is divided at the dashed line into segments (a set of connected nodes) in Fig. 2 rooted by the nodes denoted by A to I. These segments form a global tree in **Fig. 3**.

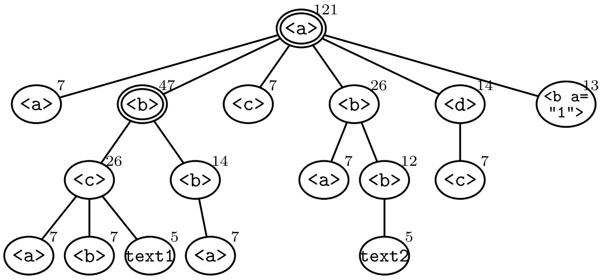

Fig. 1 Example XML tree. Numbers at upper right on nodes denote sizes of subtrees rooted by nodes. Doubled circles denote *m*-critical nodes for $m = 35$.

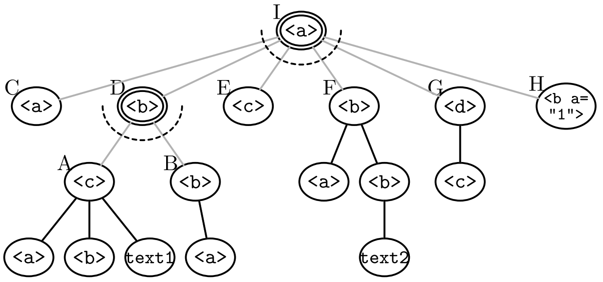

Fig. 2 Tree division immediately beneath *m*-critical nodes in Fig. 1.

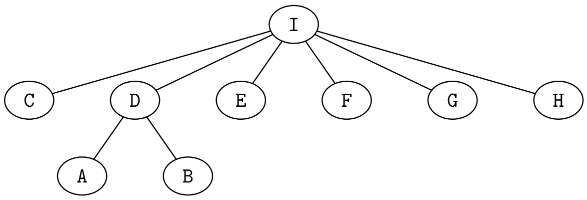

Fig. 3 Global tree corresponding to tree division in Fig. 2.

The tree division with *m*-bridges has the following properties. We have omitted the proofs from the lemmas below, since they are given in Refs. [14], [27].

Lemma 1 For each segment given by the tree division with *m*-bridges, the sum of the sizes of nodes is at most m. Here, we have excluded nodes larger than m. □

Lemma 2 When we divide a tree of N nodes with *m*-bridges, the number of *m*-critical nodes is no more than $2N/m - 1$. □

Lemma 3 Each segment given by tree division with *m*-bridges has at most one *m*-critical node. □

Lemma 1 shows that we can limit the size of each segment

[*2] The definition of *m*-bridges in this paper is a bit different from that given in Refs. [14], [27]. An *m*-critical node is included in multiple segments by the tree division in Refs. [14], [27], but it is only included in the parent segment in this paper.

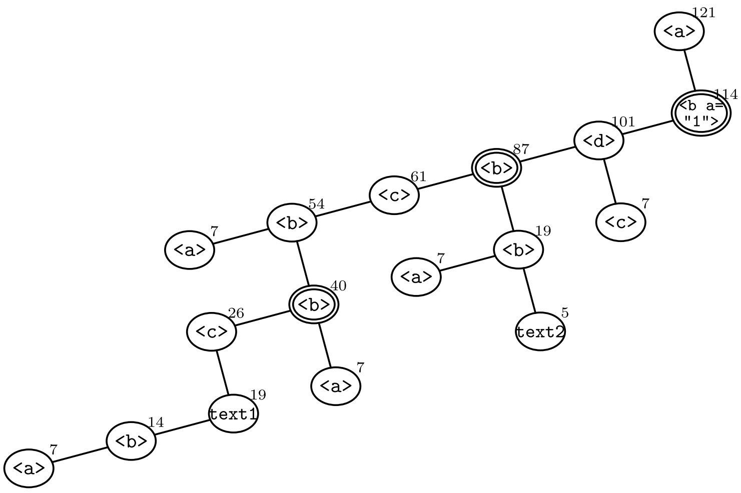

Fig. 4 Finding *m*-critical nodes for binary-tree representation of tree in Fig. 1. Numbers at upper right on nodes denote sizes of subtrees rooted by nodes in binary-tree representation. Doubled circles denote *m*-critical nodes for $m = 35$.

by m. On the other hand, the number of segments is limited by $(2N/m - 1)d + 1$ from Lemma 2 where d is the maximum number of children of a node. As is often the case with XML trees, d becomes large for shallow and wide trees and the number of segments also increases.

When the number of segments is too large, we have significant overheads in the distributed computing with MapReduce and related models. We may resolve this problem by merging segments given with m-bridges and reducing the number of divisions: e.g., a method of merging the sibling segments under an m-critical node is presented in Refs. [14], [27].

Lemma 3 is an important property for parallel computation over divided data. We will discuss this in Section 6.

3. Tree Division with *m*-bridges over One-to-one Corresponding Binary Tree

The existing tree division with m-bridges in Section 2 has a problem in that it yields too many segments particularly for shallow trees. This section explains our computation of m-bridges on a binary tree that has a one-to-one correspondence to the input tree. By computing m-bridges over the binary-tree representation, we find the resulting parts of the division are similar to the segments merged among siblings after the existing division with m-bridges.

3.1 Binary-tree Representation

First, we specify the binary-tree representation used in this study, which has one-to-one correspondence to a tree of any shape. We have presented the binary-tree representation in **Fig. 4** as an example for the tree in Fig. 1. Note that the m-critical nodes denoted by doubled circles do not match.

Definition 4 (Binary-Tree Representation) For the binary-tree representation used in this study, the nodes in the binary-tree representation and those in the original tree have the following relation. We denote a node in original tree x and the corresponding node in binary-tree representation x'.

- If node x is the rightmost child of p in the original tree, then node x' is the right child of p' in the binary-tree representation.
- If node x is the immediate left sibling of y in the original tree, then node x' is the left child of y' in the binary-tree representation. □

The parent-child relation in the original tree is a rightward parent-child relation in the binary-tree representation. The sibling relation in the original tree is a leftward parent-child relation in the binary-tree representation. Therefore, when the original tree is a shallow and wide tree, its binary-tree representation is a tall tree (in the direction to the left down). Note that the computation of m-bridges is independent of the shape of the tree, and there are no problems with how tall the binary-tree representation is.

This binary-tree representation is not the same as the well-known *left-child right-sibling representation* [9], but is its left-right reversal. We need the sizes of subtrees to determine whether a node is m-critical in tree division with m-bridges, and the sizes are computed in a bottom-up manner. The left node among siblings in the original tree is located near the root in the left-child right-sibling representation. This means that the order in the XML text representation and the order in the bottom-up computation are opposite, and we need to store all the intermediate results among sibling. These orders are the same in the binary-tree representation defined in this section, and we can reduce the amount of intermediate data (by writing out the fixed results to files).

3.2 Tree Division with *m*-bridges in Binary-Tree Representation

We divide the tree based on m-critical nodes computed on the binary-tree representation above.

We do the same computation for m-critical nodes as that with the existing algorithm. Here, note that the size of a subtree rooted at node x in the binary-tree representation is the sum of the following: the size of a subtree rooted at node x in the original tree, and the sizes of subtrees rooted at left-sibling nodes of x. As an example, the size of the subtree in Fig. 4 rooted at lower node <b>

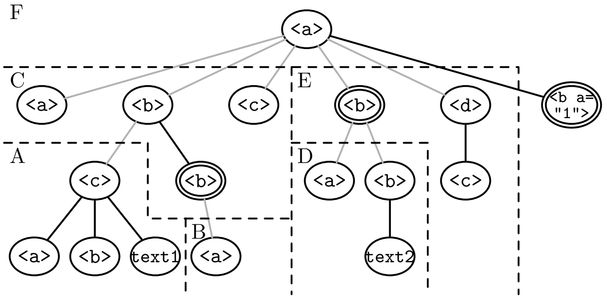

Fig. 5 Tree divided with m-critical nodes in binary-tree representation in Fig. 4.

denoted by a doubled circle is the sum of lengths 14 (for the subtree rooted at the node, <b><a></a></b>) and 26 (for the subtree rooted at the left sibling, <c><a></a><b></b>text1</c>), which is equal to 40.

We then divide the tree based on m-critical nodes. Here, we divide the tree as we do immediately beneath m-critical nodes in the binary-tree representation. Cutting a lower-right edge of a node in the binary-tree representation means cutting all the parent-child relations in the original tree. Cutting a lower-left edge of a node in the binary-tree representation means cutting the sibling relation in the original tree. Therefore, for each m-critical node, we cut the sibling relation on the left as well as the parent-child relations. We have illustrated tree division with m-critical nodes computed on the binary-tree representation in **Fig. 5**.

After tree division with m-bridges in the binary-tree representation, some parts may not be connected components. More precisely, it consists of a set of trees whose root nodes are siblings of each other. Since we call the data structure that consists of multiple trees a *forest*, after this we will also call the parts obtained after tree division forests. In the forests given by the tree division, m-critical nodes have the following properties.

Lemma 4 For each forest given by tree division with m-bridges in the binary-tree representation, there is at most one m-critical node.

Proof A forest corresponds to a segment in the binary-tree representation. It follows from the fact that Lemma 3 holds for the segment. □

Lemma 5 If a forest given by tree division with m-bridges in the binary-tree representation includes an m-critical node, then the m-critical node is the leftmost one among siblings and has no children (i.e., it is a leaf).

Proof The m-critical node is a leaf in the corresponding segment in the binary-tree representation. Since it has no child on the left in the binary-tree representation, it has no siblings on the left in the original tree. Since it has no child on the right in the binary-tree representation, it has no children in the original tree, either. □

In the same manner as we did in the existing tree division with m-bridges, we can form a global tree by reducing each segment into a node in the binary-tree representation. For the tree division in Fig. 5, we present the global tree in **Fig. 6**. Note that the global tree is a binary tree that is given from the binary-tree representation. Since m-critical nodes may not be on the uppermost layer as we can see from forest C in Fig. 5, it is not appropriate to represent a global tree in a similar manner as the original tree.

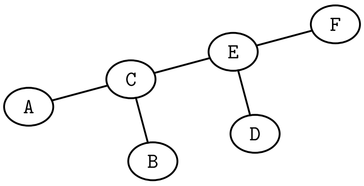

Fig. 6 Global tree corresponding to tree division in Fig. 5.

3.3 Properties on Divided Tree

In addition to the properties given in Section 2, tree division with m-bridges in the binary-tree representation has the following property.

Lemma 6 When we divide a tree of N nodes with m-bridges *in the binary-tree representation*, the number of forests is no more than $4N/m - 1$.

Proof For each m-critical node, the number of forests increases by at most two. From Lemma 2, the number of forests is at most $1 + 2 \times (2N/m - 1) = 4N/m - 1$. □

Tree division with m-bridges in the binary-tree representation guarantee the number of parts as well as the size of each part, which is different from existing tree division with m-bridges.

4. Implementation with SAX

We implemented existing tree division and proposed tree divisions with m-bridges in the binary-tree representation in this study using a Simple API for XML (SAX) library [5] and Java.

4.1 SAX

Simple API for XML (SAX) [5] is an XML parser library that provides a set of event-driven APIs. It reads an XML document from its head, and triggers events according to the types of elements. SAX is different from the Document Object Model (DOM) that puts the whole tree structure on the memory, and is not suitable for processes requiring random access to elements, but it has strong advantages that it can perform computations with a rather small memory in streaming processing. Since we assumed a very large XML as a target whose whole structure cannot be read on the memory, we developed a one-pass algorithm using the SAX library.

We need to implement a handler for each event in programming with the SAX library. The five handlers to be implemented are listed below. The tag name and attributes or text in the SAX library provided by `org.xml.sax` are passed as parameters for the events of opening tag, closing tag, or text element. We denote `p` for the position of the tag or text, and `tag` for the content of the tag name or text, for the parameter of the event handlers.

- Start of document (`startDocument`)
- Opening tag (`startElement(p, tag)`)
- Closing tag (`endElement(p, tag)`)
- Text element (`characters(p, tag)`)
- End of document (`endDocument`)

4.2 Implementation of Existing Tree Division with m-Bridges

We will first discuss implementation using the SAX library for

```
procedure startElement(p, tag)
  ys <- stack.pop()
  yL <- ys.last()
  o <- Node(pos = p,
            text = tag,
            size = tag.length,
            sum = yL.sum
            max = yL.max)
  ys.add(o)
  stack.push(ys)
  stack.push(empty_list)
end
```

Fig. 7 Event handler for opening tag in implementation of existing *m*-bridge tree division.

existing tree division with *m*-bridges discussed in Section 2.

We represent an opening tag, a closing tag, or text in the implementation by an instance of the `Node` class that has seven attributes.

- `pos`: The position of the tag or text.
- `text`: The string that represents the tag or the text.
- `size`: The size of the subtree (the length of `text`).
- `sum`: The sum of sizes of the siblings on the left.
- `max`: The maximum size of the siblings on the left.
- `isMC`: Is the node an *m*-critical node?
- `children`: The list of child tags or texts.

We can implement tree division with *m*-bridges using a stack, whose elements are vectors (variable-length lists) of the `Node` instances. The following explains the implementation of the handlers of the SAX parser.

Opening Tag

Figure 7 shows the procedure when an opening tag is read. (Exceptional processing for the case of no elements in the vector is omitted in the pseudo-code.)

When an opening tag is read, we add a new instance of the `Node` class to the top list `ys` of the stack. After the opening tag is read, the depth of nodes increases by one, and thus we add an empty list to the stack.

Closing Tag

Figure 8 shows the procedure when a closing tag is read. (Exceptional processing for the case of no elements in the vector is omitted in the pseudo-code.) This handler for a closing tag determines whether the read node is *m*-critical and outputs the segments.

When a closing tag is read, we wind back the stack by one and obtain the list of children. Then, we compute the size of the subtree including the sum of the size of the child subtrees, and add a new instance of the `Node` class to the top list `ys` of the stack. Finally, we determine whether the node and its parent node are *m*-critical, and if they are *m*-critical nodes we then output the children and siblings on the left as resulting segments, respectively. The function, `outputTrees`, in Fig. 8 is the function that outputs the segments. The function `outputTrees` takes a list of subtrees (represented by pairs of an opening tag and a closing tag) as its input, and outputs each subtree to a file with a sequence number in XML format.

```
procedure endElement(p, tag)
  cs <- stack.pop();   cL <- cs.last()
  ys <- stack.pop();   o <- ys.last()

  c <- Node(pos = p,
            text = tag,
            size = o.size + cL.sum + tag.length,
            sum = o.sum + size,
            max = max(o.max, size),
            children = cs)

  if ( isCritical(c.size, cL.max) )
    c.isMC <- true
    outputTrees(cs)
    c.children.clear()
  endif

  if ( isCritical(c.sum, c.max) )
    outputTrees(ys)
    ys.clear();  ys.add(o)
  endif

  ys.add(c)
  stack.push(ys)
end
```

Fig. 8 Event handler for closing tag in implementation of existing *m*-bridge tree division.

The function `isCritical` that determines whether the node itself and its parent are *m*-critical is defined as

$$\texttt{isCritical}(size, cmax) = (\lceil size/m \rceil > \lceil cmax/m \rceil).$$

Note that we pass the sum of sizes of sibling subtrees to the first argument when we determine whether the parent node is *m*-critical. With this assessment of whether the parent is *m*-critical, we can output some segments before reading all the siblings, which helps us to reduce memory usage. This improvement is particularly effective for a wide XML tree.

Text

We can handle a text as a consecutive pair of an opening tag and a closing tag. Here, since a text has no children, we can omit the addition of an empty list to the stack and determine whether the text itself is *m*-critical.

End of Document

We output the segment including the root node that remains on top of the stack as clean up. We also output the global tree.

The procedure for computing the global tree is as follows. When we output each segment, we store a triple (`GNode`) that consists of the ID (`NID`), the first position (`spos`), and the last position of the segment. We sort the list of these triples at the end of the document and reconstruct the global tree. The pseudo-code for computing the global tree is shown in **Fig. 9**. The argument `gnlist` is a list of `GNode` triples, sorted by `spos` in increasing order.

Output Format

A segment given by tree division may only consist of text data, and such a segment is beyond the definition of XML. To enable us to use existing XML processors for such a segment, we output the

```
procedure getGTree(gnlist)
  n <- gnlist.get(0)
  output the opening tag for n
  stack.push(n)

  for (i <- 1; i < gnlist.length; i <- i+1)
    n <- gnlist.get(i)
    while (stack.peek().epos < n.spos)
      o <- stack.pop(); output the closing tag for o
    endwhile
    output the opening tag for n
    stack.push(n)
  endfor

  while (!stack.isEmpty())
    o <- stack.pop(); output the closing tag for o
  endwhile
end
```

Fig. 9 Procedure to obtain global tree in existing *m*-bridge tree division.

segment after enclosing it with <TEXTONLY> and </TEXTONLY>.

The *m*-critical node that is included (at most one) in a segment is important information. Therefore, we add attribute `mCritical="true"` to the *m*-critical node when we output segments.

4.3 Tree division with *m*-Bridges on Binary-Tree Representation

Now, we will discuss implementation using the SAX library of tree division with *m*-bridges in the binary-tree representation proposed in Section 3. The implementation is almost the same as that of existing tree division with *m*-bridges. We use the same class `Node` (we did not use `max` in the implementation that follows).

Opening Tag

The procedure for an opening tag is the same as that in Fig. 7. Note that the size of the subtree in the binary-tree representation is given by `sum`.

Closing Tag

Figure 10 shows the procedure when a closing tag is read. (Exceptional processing for cases with no elements or a shortage of elements in the vector is omitted in the pseudo-code.) This handler for a closing tag determines whether the read node is *m*-critical and outputs forests.

```
procedure endElement(p, tag)
  cs <- stack.pop();   cL <- cs.last()
  ys <- stack.pop();   o  <- ys.last();
                       yL <- ys.last2()

  c <- Node(pos = p,
            text = tag,
            size = o.size + cL.sum + tag.length,
            sum = o.sum + size,
            children = cs)

  if ( isCritical(c.sum, cL.sum) &&
       isCritical(c.sum, yL.sum) )
    c.isMC <- true
    outputForest(cs); cs.clear();
    outputForest(ys); ys.clear();
    ys.add(o)
  endif

  ys.add(c)
  stack.push(ys)
end
```

Fig. 10 Event handler for closing tag in implementation of *m*-bridge in binary-tree representation.

```
procedure getGTree(gnlist)
  n <- gnlist.get(0)
  output the opening tag for n
  stack.push(n)

  for (i <- 1; i < gnlist.length; i <- i+1)
    n <- gnlist.get(i)
    while (stack.peek().cpos < n.epos)
      o <- stack.pop(); output the closing tag for o
    endwhile
    output the opening tag for n
    stack.push(n)
  endfor

  while (!stack.isEmpty())
    o <- stack.pop(); output the closing tag for o
  endwhile
end
```

Fig. 11 Procedure to obtain global tree in *m*-bridge in binary-tree representation.

This handler is also almost the same as that for existing tree division (Fig. 8), and thus we only focus on the differences from that. First, we obtain the sibling on the left and the rightmost child, which are two children in the binary-tree representation, from the lists in the stack. We apply the assessments of whether the node is *m*-critical to both nodes. We can use the same function for the assessments, but note that we should pass the size of the subtree in the binary-tree representation, `sum`, as the argument of the function. If the node is an *m*-critical node, we output the forests for the siblings on the left and children. The function, `outputForest`, in Fig. 10 outputs forests that correspond to segments in the binary-tree representation. The function `outputForest` takes a list of subtrees (represented as pairs of an opening tag and closing tag), and outputs them in XML format into a file whose name includes a sequence number.

Text

We can handle a text as a consecutive pair of an opening tag and a closing tag. Here, we can omit the addition of an empty list to the stack since a text has no children. However, it can be an *m*-critical node in the binary-tree representation and thus we cannot omit the assessment of whether the text itself is *m*-critical.

End of Document

We output the forest including the root node that remains on top of the stack as clean up. We also output the global tree.

The procedure for computing the global tree is as follows. When we output each forest, we store a tuple of four elements

Table 1 Experiment data.

Data name	Acquisition	height	No. of characters
DBLP.xml	DBLP Computer Science Bibliography	6	1.21×10^9
XMark2.xml	XMLCompBench http://xmlcompbench.sourceforge.net/Dataset.html	12	1.16×10^8
Random-R3.xml	XMLCompBench http://xmlcompbench.sourceforge.net/Dataset.html	30	1.32×10^7
f8i.xml	XMLGEN (http://www.xml-benchmark.org/generator.html) with parameter -f 8	13	1.01×10^9

Table 2 Tree division time (in milliseconds) and number of segments by existing *m*-bridge division.

Data	Input	Compute	Output	Total	No. of segments
DBLP.xml	10,246	30,131	126,591	166,968	3,619,811
XMark2.xml	697	2,041	2,623	5,361	70,013
Random-R3.xml	366	1,398	496	2,260	911
f8i.xml	4,957	14,034	20,359	39,350	560,013

Table 3 Tree division time (in milliseconds) and number of segments by the *m*-bridge in binary-tree representation.

Data	Input	Compute	Output	Total	No. of segments
DBLP.xml	10,246	35,284	13,038	58,568	202
XMark2.xml	697	2,301	818	3,816	214
Random-R3.xml	366	1,500	322	2,188	258
f8i.xml	4,957	15,550	5,189	25,696	210

(`GNode`) that consists of the ID (`NID`), the first position (`spos`), the last position (`epos`), and the position of the *m*-critical node (`cpos`) of the forest. At the end of the document, we sort the list of these tuples and reconstruct the global tree. The pseudo-code for computing the global tree is given in **Fig. 11**. The argument `gnlist` is a list of `GNode` tuples, sorted by `epos` in decreasing order.

Output Format

Each part is a forest after tree division with *m*-bridges in the binary-tree representation. Therefore, we output the forest in XML format after enclosing it with a pair of dummy tags. We deal with *m*-critical nodes in the same way as what we discussed for existing tree division with *m*-bridges.

5. Experiments

We evaluated the execution time and the number of segments or forests with implementations using the SAX library discussed in Section 4 through experiments for existing tree division with *m*-bridges and the proposed tree division with *m*-bridges in the binary-tree representation.

The hardware we used for the experiments had a CPU Core i7-4770, 32 GB of memory, and solid state drive (SSD) was the storage device. The software we used was Ubuntu 13.04 and Java8.

5.1 Execution Times

We compared the execution times of existing tree division with *m*-bridges and the proposed tree division with *m*-bridges in the binary-tree representation. We used the test data listed in **Table 1**. Parameter *m* was set by using one-hundredth the number of characters.

Tables 2 and **3** and **Fig. 12** present the experimental results. The execution times are the average of five executions. We have plotted the execution time per character in Fig. 12 in units of ms/MB. The breakdown is categorized into three functions.

- Read: The time consumed by the SAX parser to read the XML file and call the handler functions. We measured this time by executing a program with empty handler functions.
- Compute: The main computation time of tree division that excludes the input and output of data. We calculated this time by subtracting the read/write times from the whole execution time.
- Write: The time for outputting the segments or forests and the global tree to the files. We measured the execution time with a program that output nothing and calculated the difference from the whole execution time.

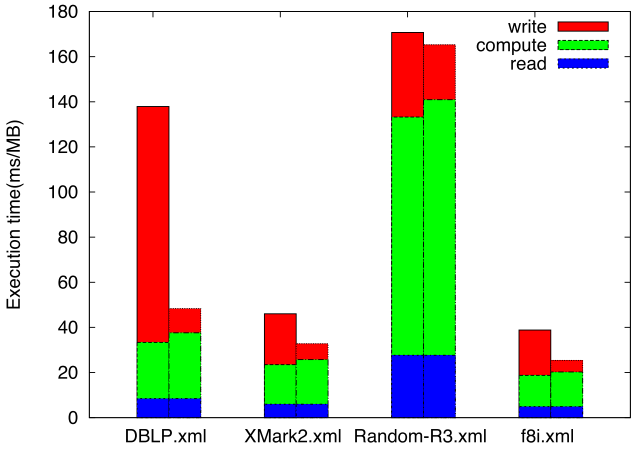

Fig. 12 Execution times for existing *m*-bridge division and *m*-bridge in binary-tree representation for 1 MB input. Bars at left indicate results for existing *m*-bridge division and those at right indicate *m*-bridge in binary-tree representation.

The whole execution times in Fig. 12 indicated that the proposed tree division with *m*-bridges in the binary-tree representation processes data faster than the existing tree division with *m*-bridges. Looking at the execution times in detail, while the main computation time of the proposed tree division with *m*-bridges in the binary-tree representation is a bit larger than that of the existing tree division with *m*-bridges, the writing time by the former is much smaller than that by the latter. These results are most notable for DBLP.xml, and thus we can see that tree division with *m*-bridges in the binary-tree representation achieves better speedup

Table 4 Execution times for DBLP.xml (in milliseconds).

	Existing	Proposed
(A) Original program	168,444	12,273
(B) No global tree	163,430	13,366
(C) Same output filename	130,239	17,815
(D) Appending in same file	10,482	16,083

for shallower and wider trees.

To check the details, we obtained a breakdown of the execution times of the existing and proposed tree divisions for DBLP.xml. We used four programs.

- (A) Used the original program.
- (B) Omitted the computation and output of the global tree.
- (C) (B) + Used the same filename for the outputs.
- (D) (B) + Appended the outputs in the same file.

We could compute the time for computing the global tree and outputting it from the difference between (A) and (B). We could find the overheads caused by too many output files from the differences between (B), (C), and (D). In particular, the differences between (C) and (D) revealed the overheads caused by the opening/closing of files, and the differences between (B) and (C) revealed the OS-level overheads. We have summarized the writing times of the programs in **Table 4**[*3]. These results indicate the time for computing and outputting the global tree is about 5 seconds in existing tree division. We can also see the main reason for the long writing time is the overhead caused by too many output files.

The execution time per character is longer in Random-R3.xml than that for the other data. Random-R3.xml has too many tags compared with the data size, and the handler functions were called once per 4.4 characters on average. We considered that the reason for the long execution time was this cost of calling handler functions.

The numbers of divisions in Tables 2 and 3 indicate that while the existing tree division with m-bridges has very many divisions, the number of divisions in the proposed tree division with m-bridges in the binary-tree representation is stable over the data. This implies that we can easily set value m appropriately according to the required number of divisions for tree division with m-bridges in the binary-tree representation.

5.2 Properties of Tree Division

We investigated the distribution of the sizes of segments or forests after existing tree division with m-bridges and the proposed tree division with m-bridges in the binary-tree representation. We used a set of randomly-generated trees in this experiment, which was generated by a program developed by the authors. We set the number of nodes at 10,000,000 (about 100 MB) and the average branch factors at 4 and 20. We generated 10 trees with different random seeds for each set of parameters. We divided the tree by using the existing and proposed methods with m-bridges with parameter m = 100,000, and we plotted the sizes of segments or forests in cumulative frequency graphs. The results were averaged for 10 trees.

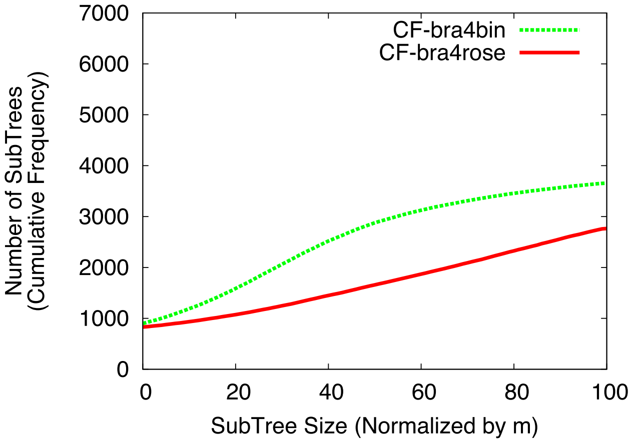

Fig. 13 Cumulative frequency of segment sizes for input tree with 10,000,000 nodes (size approx. 100 MB) and an average branch factor of 4, with m = 100,000.

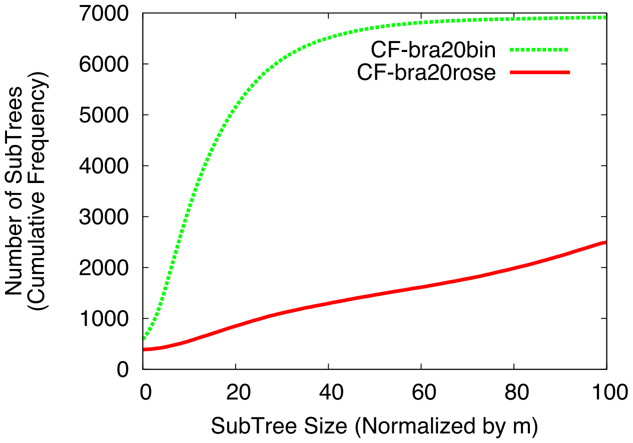

Fig. 14 Cumulative frequency of segment sizes for input tree with 10,000,000 nodes (size approx. 100 MB) and an average branch factor of 20, with m = 100,000.

Figures 13 and **14** present the experimental results. The horizontal axes plot the percentage for m = 100,000. The results for the average branch factor of 4 in Fig. 13 indicate that the number of divisions with the existing method is larger than that with the proposed method, particularly for segments with sizes under 40% of m. The average branch factor of 20 in Fig. 14 indicates that there are too many small segments in existing tree division with m-bridges compared with the proposed tree division with m-bridges in the binary-tree representation. Since the trees were wide, a node with many children became an m-critical node yielding a lot of small segments after division. These results reveal that the existing tree division with m-bridges is affected by the shape of trees. However, the plots for the proposed tree division with m-bridges in the binary-tree representation are similar for the average branch factors of 4 and 20, and we can see that it is stable against the shapes of trees.

6. Application to Distributed Computing

This section discusses how we can use the data, which are divided by tree division with m-bridges in the binary-tree representation proposed in Section 3, in distributed computing.

[*3] The execution times were counter to our expectations for the proposed tree division. We could not clarify what the reasons were.

6.1 Expected Usage

Expected usage could be tree processing with the MapReduce framework [10] in cloud environments.

We generally need to upload data before we use cloud environments. Here, when users upload tree structures like XML, it is useful for them if the cloud systems automatically divide the trees based on the tree division proposed in this study. We developed one-pass tree division algorithms with this application in mind. In fact, of the experimental results in Fig. 12, all the results except for Random-R3.xml indicate that we can divide trees with a speed of more than 20 MB/s, and the authors consider this is fast enough compared with the network speed in cloud environments. Therefore, users can upload tree-structured data without any overheads, and then execute MapReduce programs for the divided tree data.

6.2 A Query Example for Divided Data

Since tree division with m-bridges yields independent segments (or forests), we can compute them independently in parallel. When we utilize the MapReduce framework [10], we can compute partial results independently on each set of nodes in the map phase, and then merge the partial results based on the global tree in the reduce phase.

Here, we present the idea of MapReduce computation of an XPath query `//b[following-sibling::d && child::c]` for the divided data in Figs. 5 and 6. This XPath query points to node `b` that has at least one child `c` and at least one sibling `d` on the right.

We independently compute values or functions for each forest in the map phase. For a forest that corresponds to a leaf in the global tree, we compute a value as the result; for a forest that corresponds to an internal node in the global tree, we compute a function that takes values from the sibling on the left and from the children.

The value computed from forest x for our running example is either a triple (x_c, x_b, x_d) defined as follows or a function that returns the triple. The values in the triple have the three meanings.

- x_c: Is there a node `c` in the top layer of the forest?
- x_b: Is there a node `c` that has at least one child `c` in the top layer of the forest?
- x_d: Is there a node that satisfies the whole XPath query?

We denote the function that computes these values or functions as f. The results are given as follows. Here, T and F represent true and false.

$$f(A) = (T, F, F)$$
$$f(B) = (F, F, F)$$
$$f(C) = \lambda(l_c, l_b, l_d)(c_c, c_b, c_d).(T, l_c, l_d \vee c_d)$$
$$f(D) = (F, F, F)$$
$$f(E) = \lambda(l_c, l_b, l_d)(c_c, c_b, c_d).(l_c, l_b \vee c_c, l_d \vee c_d \vee l_b)$$
$$f(F) = \lambda(l_c, l_b, l_d)(c_c, c_b, c_d).(F, F, l_d \vee c_d)$$

Here, the property in Lemma 4 that there is at most one m-critical node in the forest plays an important role. If a forest includes multiple m-critical nodes, then function f should take more than two parameters. Since a forest includes at most one m-critical node, we can guarantee that function f returns either a value or a binary function.

We finally compute the final results by assigning these results to the global tree in Fig. 6 and reducing them in a bottom-up manner. We assume value (F, F, F) for the missing child in the global tree (e.g., the root node in Fig. 6). We can compute the value for C as (T, T, F), then for E as (T, T, T), and finally for F as (F, F, T) in our running example. The final result of the query is given as the third element, which is T.

7. Related Work

7.1 Parallel Processing on Tree Structures

Tree contraction algorithms [1], [24], [27], which apply tree-contraction operators in parallel to a set of independent local nodes, are important parallel algorithms for tree structures. Tree division with m-bridges was first proposed within the context of these tree contraction algorithms [14], [27].

Tree division has been performed in shared-memory environments in Refs. [14], [27]. It is sufficient to enumerate m-critical nodes in shared-memory environments, and an algorithm for enumeration has been proposed based on the list-ranking algorithm on an Euler tour of trees. Since the text representation of XML corresponds to the Euler tour of trees, the SAX-based algorithm for the existing tree division in this paper is almost the same as the algorithm in Refs. [14], [27]. We devised the algorithm so that we could write out the m-bridges in one pass while limiting memory usage. The authors [14], [27] also discussed how we could deal with trees of unbound degree: in particular, the algorithm divided trees with m-bridges and then merged at most n segments among siblings.

Gibbons et al. [15] and Skillicorn [29] abstracted tree contraction algorithms based on a functional programming approach, and proposed a set of parallel tree-computational patterns called parallel tree skeletons. Since the concrete order of computation is hidden behind the parallel tree skeletons, users can perform parallel computation just by passing the operators used in the computation. The authors developed a parallel skeleton library called *SkeTo* [12], [22], and it included efficient parallel tree skeletons for binary trees [20] and even for general trees [23]. In particular, we [23] implemented parallel general-tree skeletons based on a left-child right-sibling binary-tree representation and parallel binary-tree skeletons. The left-right relation among siblings is not essential in computational patterns [23], and thus we can implement them in the same way on the binary-tree representation explained in this paper. Compared with the implementation in Ref. [23], the proposed tree division has an advantage in that it does not require any dummy nodes in the binary-tree representation.

Another approach to parallel computation on tree structures is to apply *flattening* [18] to transform a tree into a one-dimensional array. The main advantage of flattening transformation is that we can easily divide the flattened array. We can consider the text representation of XML as an flattened tree. Parallel algorithms for these flattened trees were proposed by Sevilgen et al. [28] and by Kakehi et al. [16]. We place the intermediate results on the stack in these algorithms and these results are communicated among processors. Therefore, the MapReduce implementation [11] of the tree reduction algorithm by Kakehi et al. was more complicated than the MapReduce algorithm discussed in this study.

7.2 Distributed Queries for Tree Structure and Tree Division

The methods of distributing XML data and conducting distributed queries on them have been studied intensively in the XML-database community.

Bremer et al. [4] proposed a method of data-distribution with the RepositoryGuide, where each part of data consists of a common path from the root to a node and the subtrees rooted at the node. If the data are distributed in this manner, we can rather easily conduct distributed queries that retrieve a node from the root. Ma et al. [19] categorized the distribution of XML databases from the viewpoint of object-oriented databases into three: split, horizontal, and vertical.

We can categorize studies on distributed queries into two groups in terms of data division. The first is about distributed queries that deal with divided data evenly, and these studies have been well summarized by Morihata [25]. Suciu [30] proposed a method of conducting path queries that were specified with automata on the divided graph structures. Nomura et al. and one of the authors [21], [26] proposed a method of implementing XPath queries with predicates using parallel tree skeletons. Cong et al. formalized computation with a partial-evaluation technique [6], [8].

The second group involves distributed queries for data distributed in similarly to those by as Bremer *et al*. These queries basically consist of two parts: the first is for the path from the root and the second is for the subtree rooted at the node. Many studies have been done on the efficient implementation of queries that scan the tree from the root [2], [3], [13]. For example, Bordawekar et al. [2], [3] proposed a method of conducting queries in parallel based on the division of queries as well as the division of XML data. Toyonaga [31] evaluated its performance and improved it. Parallel queries that utilize paths from the root to some nodes have also been implemented in HadoopXML [7].

8. Conclusion

Data division is the first step in the distributed computing. We improved the existing method of tree division with *m*-bridges in this study and tree division with the proposed method was fast and stable for general XML trees. We also implemented algorithms using the SAX library and confirmed that we could divide trees sufficiently fast. We discussed the use of data divided with the proposed method for distributed computing using an XPath query as an example.

The proposed tree division with *m*-bridges in binary-tree representation has two main advantages.

- The number of parts yielded by division is small and is stable against the shapes of the trees.
- Since the number of parts is small, the total time for tree division is short.

Our future work includes implementing query applications and designing a MapReduce framework over divided trees.

Acknowledgments Part of this work was supported by JST/ANR Grants Numbered 10102704/ANR-2010-INTB-0205-02.

References

[1] Abrahamson, K.R., Dadoun, N., Kirkpatrick, D.G. and Przytycka, T.M.: A Simple Parallel Tree Contraction Algorithm, *Journal of Algorithms*, Vol.10, No.2, pp.287–302 (1989).

[2] Bordawekar, R., Lim, L., Kementsietsidis, A. and Kok, B.W.-L.: Statistics-based parallelization of XPath queries in shared memory systems, *EDBT 2010, Proc. 13th International Conference on Extending Database Technology*, pp.159–170, ACM (2010).

[3] Bordawekar, R., Lim, L. and Shmueli, O.: Parallelization of XPath queries using multi-core processors: Challenges and experiences, *EDBT 2009, Proc. 12th International Conference on Extending Database Technology*, pp.180–191, ACM (2009).

[4] Bremer, J.-M. and Gertz, M.: On Distributing XML Repositories, *International Workshop on Web and Databases*, pp.73–78 (2003).

[5] Brownell, D.: *SAX2*, O'Reilly Media (2002).

[6] Buneman, P., Cong, G., Fan, W. and Kementsietsidis, A.: Using Partial Evaluation in Distributed Query Evaluation, *Proc. 32nd International Conference on Very Large Data Bases*, pp.211–222, ACM (2006).

[7] Choi, H., Lee, K.-H., Kim, S.-H., Lee, Y.-J. and Moon, B.: HadoopXML: A Suite for Parallel Processing of Massive XML Data with Multiple Twig Pattern Queries, *Proc. 21st ACM International Conference on Information and Knowledge Management* (*CIKM'12*), pp.2737–2739, ACM (2012).

[8] Cong, G., Fan, W., Kementsietsidis, A., Li, J. and Liu, X.: Partial Evaluation for Distributed XPath Query Processing and Beyond, *ACM Trans. Database Syst.*, Vol.37, No.4, pp.32:1–32:43 (2012).

[9] Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C.: *Introduction to Algorithms*, MIT Press, 3rd edition (2009).

[10] Dean, J. and Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters, *6th Symposium on Operating System Design and Implementation* (*OSDI2004*), pp.137–150 (2004).

[11] Emoto, K. and Imachi, H.: Parallel Tree Reduction on MapReduce, *Proc. International Conference on Computational Science, ICCS 2012*, Procedia Computer Science, Vol.9, pp.1827–1836, Elsevier (2012).

[12] Emoto, K. and Matsuzaki, K.: An Automatic Fusion Mechanism for Variable-Length List Skeletons in SkeTo, *International Journal of Parallel Programming* (2013). Available online.

[13] Figueiredo, G., Braganholo, V. and Mattoso, M.: Processing Queries over Distributed XML Databases, *Journal of Information and Data Management*, Vol.1, No.3, pp.455–470 (2010).

[14] Gazit, H., Miller, G.L. and Teng, S.-H.: Optimal Tree Contraction in EREW Model, *Proc. Princeton Workshop on Algorithms, Architectures, and Technical Issues for Models of Concurrent Computation*, pp.139–156 (1987).

[15] Gibbons, J., Cai, W. and Skillicorn, D.B.: Efficient Parallel Algorithms for Tree Accumulations, *Science of Computer Programming*, Vol.23, No.1, pp.1–18 (1994).

[16] Kakehi, K., Matsuzaki, K. and Emoto, K.: Efficient Parallel Tree Reductions on Distributed Memory Environments, *ICCS 2007: Proc. 7th International Conference, Part II*, Lecture Notes in Computer Science, Vol.4488, pp.601–608, Springer (2007).

[17] Kawamura, T. and Matsuzaki, K.: Evaluation of Tree Processing based on the m-bridge Technique over Hadoop, *Proc. 29th JSSST Conference* (2012). In Japanese.

[18] Keller, G. and Chakravarty, M.M.T.: Flattening Trees, *Euro-Par '98 Parallel Processing, Proc. 4th International Euro-Par Conference, 1998*, Lecture Notes in Computer Science, Vol.1470, pp.709–719, Springer (1998).

[19] Ma, H. and Schewe, K.-D.: Fragmentation of XML Documents, *Journal of Information and Data Management*, Vol.1, No.1, p.21 (2010).

[20] Matsuzaki, K.: Efficient Implementation of Tree Accumulations on Distributed-Memory Parallel Computers, *ICCS 2007: Proc. 7th International Conference, Part II*, Lecture Notes in Computer Science, Vol.4488, pp.609–616, Springer (2007).

[21] Matsuzaki, K.: Parallel Programming with Tree Skeletons, PhD Thesis, Graduate School of Information Science and Technology, The University of Tokyo (2007).

[22] Matsuzaki, K., Akashi, Y., Emoto, K., Iwasaki, H. and Hu, Z.: SkeTo: A Library for Parallel Programming with Constructive Skeletons, *Proc. 22nd JSSST Conference* (2005). In Japanese.

[23] Matsuzaki, K., Hu, Z. and Takeichi, M.: Parallel Skeletons for Manipulating General Trees, *Parallel Computing*, Vol.32, No.7–8, pp.590–603 (2006).

[24] Miller, G.L. and Reif, J.H.: Parallel Tree Contraction and its Application, *26th Annual Symposium on Foundations of Computer Science*, pp.478–489, IEEE Computer Society (1985).

[25] Morihata, A.: Work Efficient Distributed XPath Querying, *Proc. 30th JSSST Conference* (2013). In Japanese.

[26] Nomura, Y., Emoto, K., Matsuzaki, K., Hu, Z. and Takeichi, M.: Parallelization of XPath Queries with Tree Skeletons, *Computer Software*, Vol.24, No.3, pp.51–62 (2007). In Japanese.

[27] Reif, J.H. (Ed.): *Synthesis of Parallel Algorithms*, Morgan Kaufmann Publishers (1993).

[28] Sevilgen, F.E., Aluru, S. and Futamura, N.: Parallel algorithms for tree accumulations, *Journal of Parallel and Distributed Computing*, Vol.65, No.1, pp.85–93 (2005).

[29] Skillicorn, D.B.: Parallel Implementation of Tree Skeletons, *Journal of Parallel and Distributed Computing*, Vol.39, No.2, pp.115–125 (1996).

[30] Suciu, D.: Distributed query evaluation on semistructured data, *ACM Trans. Database Syst.*, Vol.27, No.1, pp.1–62 (2002).

[31] Toyonaga, S.: Parallelization of XML Queries by multicore-processors, Master's thesis, Graduate School of Information Science, Nara Institute of Science and Technology (2012). In Japanese.

Takayuki Kawamura received his B.E. and M.E. degrees from Kochi University of Technology in 2012 and 2014, respectively.

Kiminori Matsuzaki is an Associate Professor of Kochi University of Technology in Japan. He received his B.E., M.S. and Ph.D. from The University of Tokyo in 2001, 2003 and 2007, respectively. He was an Assistant Professor (2005–2009) in The University of Tokyo, before joining Kochi University of Technology as an Associate Professor in 2009. His research interest is in parallel programming and algorithm derivation. He is a member of ACM, JSSST, IEEE.