アルゴリズム 86－3
(2002. 9. 19)

2分探索木を平衡化する並列アルゴリズム

右田 雅裕 † 多田 昭雄 †† 中村 良三 ‡
† 熊本大学 総合情報基盤センター †† 崇城大学 工学部 ‡ 熊本大学 工学部

本稿では2分探索木を平衡化する効率よい並列アルゴリズムをCREW-PRAMモデル上で提案する．まず，オイラーツアー技法を用いて2分探索木の節点の通りがけ順の値を求める効率よい並列アルゴリズムを提案する．提案するアルゴリズムは，節点数をnとすると，$O(n)$プロセッサを用いて$O(\log n)$時間で実現できる．次に，求めた通りがけ順の値を利用して，定数時間で2分探索木の平衡化を行う並列アルゴリズムを提案する．提案する並列アルゴリズムは従来のアルゴリズムと比較しプロセッサ数と記憶領域ともに減少することができる．

Parallel Algorithm for constructing an Almost Complete Balanced Binary Search Tree

Masahiro Migita† Akio Tada†† Ryozo Nakamura‡
†Center for Multimedia and Information Technologies, Kumamoto University ††Department of Electrical System and Computer Engineering, Faculty of Engineering, Sojo University ‡Department of Computer Science, Faculty of Engineering, Kumamoto University

In this paper an efficient parallel algorithm for balancing a binary search tree is proposed in CREW-PRAM model. At first, an efficient parallel algorithm to number the nodes in inorder on a binary search tree by using the Euler Tour technique is presented. The proposed algorithm can be implemented in $O(\log n)$ time with $O(n)$ processors, where n is the number of nodes in the tree. Next, an alternative parallel algorithm for constructing an almost complete balanced binary search tree in constant time is proposed, once the inorder number of each node is determined. The proposed parallel algorithm can reduce both the number of processors and the memory spaces as compared with the traditional algorithm.

1 Introduction

The binary search tree technique is one of the most fundamental and important computer algorithms. The search cost on the binary search tree is quite dependent on the shapes of the trees. In a binary search tree built from n nodes, the operations such as INSERT, DELETE and SEARCH require $O(\log n)$ units of time on the average, however, in the worst case, these operations take linear time as $O(n)$. In order to eliminate the worst case performance of the tree, several parallel balancing algorithms have been proposed and implemented[1)~4)].

In this paper, we propose two parallel algorithms. The first proposed parallel algorithm is to count the inorder number of nodes of a given binary search tree. To construct this algorithm first, we generate the Euler circuit by performing the Euler tour technique to the given adjacency list of binary search tree, then generate the traversal list. From the traversal list, we construct the parallel algorithm to get the inorder number of node of the tree. We have already published this algorithm in the technical note written in Japanese such as reference 6). The second proposed parallel algorithm is to construct an almost complete balanced binary search tree, using the inorder number determined by first proposed algorithm. Both the proposed parallel algorithms can be executed on CREW-PRAM model and the first proposed parallel algorithm can be implemented in time $O(\log n)$ with $O(n)$ processors, and the second one can be implemented in time $O(1)$ and can reduce both the number of processors and the memory spaces as compared with the traditional algorithm.

2 Parallel Algorithm for Inorder traversal on a Binary Search Tree

Proposed parallel algorithm first generates the traversal list from the Euler circuit of binary search tree constructed by the Euler tour technique, then efficiently calculates the inorder traversal number on the traversal list.

In the parallel algorithm for traversing a tree described in reference 5), first get the Euler circuit by performing the Euler tour technique to the tree represented in the form of adjacency list (in this list the edge (i, j) and the edge (j, i) is referable to each other).

The Euler tour technique is a parallel compu-

tation to construct the Euler circuit of the tree whose each branch is substituted by two edges of the mutually reverse direction, then generates the traversal list in which the root node is the head of the list from the circuit. Next we calculate the rank of each element (edge) of the traversal list. Here, the rank of an element kth of the list is defined as the number of elements from the head of the list to the element kth. As for the rank, it is known that we can calculate it on the list consisting of n elements with time complexity $O(\log n)$ using $O(n/\log n)$ processors using doubling technique on EREW-PRAM model.

After we get the rank of each element of the traversal list, an arbitrary branch (i, j) of the tree appears in the traversal list as the pair of edges (i, j) which has smaller rank value and (j, i) which has the greater rank value. Here, we call the edge having the smaller rank value as **advance edge** and that of greater rank value as **retreat edge**.

In the proposed parallel algorithm, first, we construct a traversal list from the Euler circuit of a binary search tree by the Euler tour technique as described above, then we will get the advance edge or the retreat edge of each edge of the traversal list.

For example, we represent a binary search tree, shown in Fig.1 of the data structure shown in Fig.2. Considering that this tree is ordered tree, we construct the elements of the adjacency list in the order of parent, left child and right child. Here, we understand that a value zero means the lack of the element, and the column 'num' indicates the inorder number of each node of the tree and it will be calculated later.

Next, we make the root node of the binary search tree as the head of the traversal list. Then, calculate the advance edge and retreat edge of the traversal list using the rank of each element of the list.

The proposed algorithm uses the following data structure in the shared memory area, and the binary search tree is represented by the following array T.

```
type index = 1..N;
     node  = record
                 key:keytype; { predefined }
                 num:index;
                 parent,left,right:index;
             end;
var  T = array[index] of node;
```

If the advance and retreat edges of the traversal list are known in advance, we can easily get the preorder and postorder traverses by the following

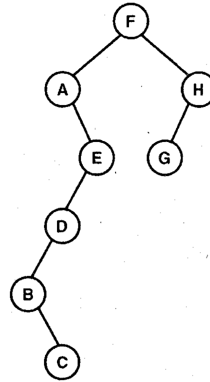


Fig. 1: Example of a binary search tree

Array T	key	num	parent	left	right
1	F	6	0	2	3
2	A	1	1	0	5
3	H	8	1	4	0
4	G	7	3	0	0
5	E	5	2	6	0
6	D	4	5	7	0
7	B	2	6	0	8
8	C	3	7	0	0

Fig. 2: Binary Search tree Representation of Fig.1

procedures[5].

(i) preorder

We construct a sublist of the elements of advance edges from the traversal list as follows.

$$(i_1, j_1) \longrightarrow (i_2, j_2) \longrightarrow \cdots \longrightarrow (i_n, j_n)$$

Head element (i_1, j_1) of the above sublist is traced first node i_1 then node j_1 and the kth element (i_k, j_k) $(k \geq 2)$ is traced only j_k. Therefore, the preorder traverse of the nodes is as follows.

$$i_1 \longrightarrow j_1 \longrightarrow j_2 \longrightarrow j_3 \longrightarrow \cdots \longrightarrow j_n$$

(ii) postorder

We construct a sublist of elements of retreat edges from the traversal list as follows.

$$(i'_1, j'_1) \longrightarrow (i'_2, j'_2) \longrightarrow \cdots \longrightarrow (i'_n, j'_n)$$

We take out an item i'_k of the edge (i'_k, j'_k) $(k = 1, 2, ..., n-1)$ from head of the sublist and take out two items i'_n, j'_n from the last element of the list. As a result tracing of the nodes becomes as follows.

$$i'_1 \longrightarrow i'_2 \longrightarrow i'_3 \longrightarrow \cdots \longrightarrow i'_n \longrightarrow j'_n$$

The main points of the above traverse (i) and (ii) has been shown in reference 5), but the inorder traverse has not been shown. Our proposed algorithm

for inorder traverse is designed in the same manner as the above mentioned algorithms (i) and (ii).

(iii) inorder

Preorder and postorder traverses are traced easily using the Euler tour technique as mentioned previously[5]. Although the inorder traverse could be calculated from the Euler tour technique but the algorithm has not proposed completely. Thus we propose a new idea to calculate the inorder traverse. By inorder means, we need to trace a node in the order left branch, a root and right branch. We conceive that the left branches are traced in postorder and the right branches in preorder. We therefore, add the information of the left branch or the right branch to the information of an advance edge or a retreat edge on the traversal list. As a result, we will propose the algorithm to trace the nodes in inorder as follows.

2.1 Proposed Parallel algorithm for inorder traversal

First we create the sublist of the elements which satisfy the conditions from (a) to (e) as given below, next we take out in the order the first node of the element (pair of nodes) of the sublist to get the inorder traversal.

We assign a processor to each edge on a traversal list in order to briefly describe the algorithm. At first we attach a mark to each element satisfied the condition of (a), (b) or (c) as follows.

- (a) Retreat edge of the left branch
- (b) Advance edge of the right branch
- (c) Retreat edge of the right branch that follows the advance edge of the right branch, or retreat edge of the right branch that follows the retreat edge of the left branch
- (d) We make the sublist consisted of the marked elements. If the last edge (i_n, j_n) of the sublist is a retreat edge of the left branch, then we add dummy edge (j_n, j_{n+1}) to the tail of the sublist
- (e) If there is an edge which follows a retreat edge of the right branch on the sublist formed with the conditions from (a) to (d), the edge (retreat edge of the left branch) is removed

Finally, we take up the first node of each edge from the sublist satisfied the condition (e) and make the list consisted of only the nodes taken up. Then we apply the parallel list ranking algorithm to the list and get the inorder number of each node.

The above algorithm is shown in below Fig.3. We assume that the tail element of the sublist points to itself and we assign the following array to each element kth of the sublist. We define $son(k)$, for identification of left branch or right branch; $trav(k)$, for identification of advance edge or retreat edge; $reverse(k)$, for pointing to edge (j, i) of reverse order of edge (i, j); $mark(k)$ for marking the indication to satisfy the condition of (a), (b) or (c). In addition the function $head(L)$ points the head element of the list L and $next(k)$ shows the pointer.

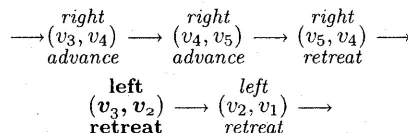
2.2 Proof of the correctness of the Proposed algorithm

First, it is self-evident that an edge satisfied the condition of (a) or (b) becomes a candidate of inorder traverse. In addition, in the condition (c), we can trace leaf node v of the right branch by including retreat edge (v, u) of the right branch which follows the advance edge (u, v) of the right branch in the sublist. We also can trace node v which does not has the right branch by including retreat edge (v, w) of the right branch which follows the retreat edge (u, v) of the left branch.

Next in the condition (d), after we make the sublist consisted only of the elements satisfying the condition of (a), (b) or (c). If the last edge (i_n, j_n) of the sublist is a retreat edge in the left branch, in other word, if root j_n does not have the right branch, we add a dummy edge (j_n, j_{n+1}) and we can trace the root j_n.

In the condition (e), if there is the edge which follows the retreat edge of the right branch in the sublist consisted of the edge satisfying the conditions from (a) to (d), its edge has to be the retreat one of the left branch. Because the edges in the right branch have already been existed on the sublist, the edge which follows the retreat edge of the right branch has to be the retreat edge on the left branch. Since node v in the retreat edge (v, u) of the left branch which just follows the retreat edge of the right branch has been existed in advance edge (v, w) of the right branch already, node v is traced twice. Therefore, a retreat edge of the left branch which follows the retreat edge of the right branch has to eliminate from the sublist.

For example, the sublist of Fig.4 is shown after condition (d) is satisfied where the retreat edge (v_3, v_2) of the left branch which follows the retreat edge (v_5, v_4) of the right branch eliminates from the sublist, as given below.



In what follows we show the performance of proposed algorithm for binary search tree in Fig.1. At

```
const n = { total number of nodes };
type cell = record  start,end:  1..n+1;
            next:  list  end;
     list = ↑cell;

procedure Para_Inorder_Traverse(traversal_list:  list);
var rank:  array of integer;  k, sublist:  list;
begin  { calculate rank of traversal list }
   List_Ranking(traversal_list, rank);
   for all k, k ∈ traversal_list in parallel do
   begin  { advance edge or retreat edge identification }
      if rank(k) < rank(reverse(k)) then
         trav(k) ← advance
      else
         trav(k) ← retreat;
      mark(k) ← 0;
      if trav(k) = retreat and son(k) = left then
         mark(k) ← 1  { condition (a) }
      else if trav(k) = advance and son(k) = right then
         mark(k) ← 2;  { condition (b) }
      if mark(k) ≠ 0 and next(k) ≠ k then
         if trav(next(k)) = retreat and
            son(next(k)) = right then
               mark(next(k)) ← 3  { condition (c) }
   end;
   sublist ← traversal_list;  { from here condition (d) }
   repeat log 2(n − 1) times
   { make a sublist consisted from only marked edges }
      for all k, k ∈ sublist in parallel do
         if mark(next(k)) = 0 then
            next(k) ← next(next(k));
   for all k, k ∈ sublist in parallel do begin
      if k=head(sublist) then
      { process for the head of the sublist }
         if mark(k) = 0 then
            head(sublist) ← next(head(sublist));
      if mark(next(k)) = 0 then
         next(k) ← k;  { process for the tail of the sublist }
      if next(k) = k and mark(k) = 1 then
      begin
      { case that the tail is a retreat edge of the left branch }
         new(next(k));  { add a dummy edge to the tail }
         next(next(k)) ← next(k);
         next(k)↑.start ← k↑.end; next(k)↑.end ← n+1
      end
   end;
   for all k, k ∈ sublist in parallel do
      if mark(k) = 3 and next(k) ≠ k then
         next(k) ← next(next(k));  { condition (e) }
   { calculation of inorder number }
   List_Ranking(sublist, rank);
   for all k, k ∈ sublist in parallel do
      T[k↑.start].num ← rank(k)  { storing the value }
end;
```

Fig. 3: Parallel algorithm for numbering the vertices in inorder using the traversal list

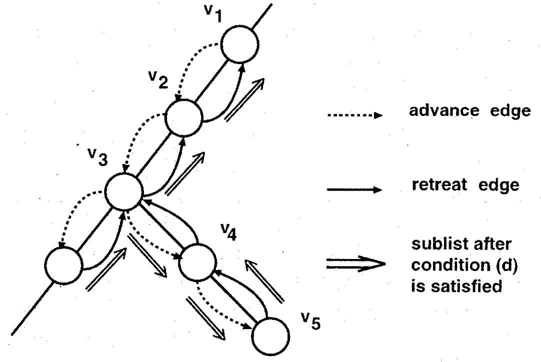
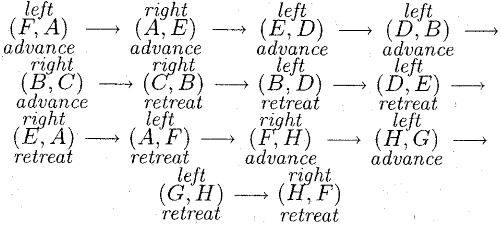
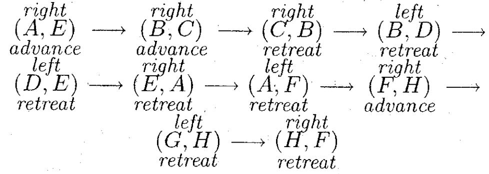


Fig. 4: Traversal list and sublist after the condition (d) is satisfied

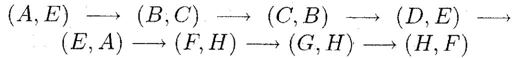
first we add marks to distinguish advance and retreat edge, right and left branch on the traversal list as follows.



Sublist to satisfy the conditions from (a) to (d) as mentioned above on the traversal list becomes as follow.



Next, applying the condition (e), the edges of (B, D) and (A, F) are eliminated and the sublist becomes as follows.



Sublist as mentioned above is shown by the directed arrows in Fig.5, the dotted and solid arrows are shown with the applied conditions. The result after the condition (e) is shown by the solid arrows, the inorder traverse is given if we take out a head node of each element of this sublist as follows $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F \rightarrow G \rightarrow H$, and apply the parallel list ranking to the above i-

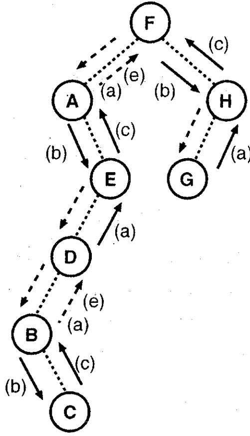


Fig. 5: Sublist satisfied the conditions $(a)\sim(e)$ in Fig.1

norder traversal list, finally we can get the inorder number.

2.3 Time complexity of the proposed algorithm

The algorithm to get the Euler circuit on a tree needs $O(\log n)$ time from the adjacency list representation of the tree with nodes n, but it can be calculated with $O(1)$ constant time in case of binary search tree, because the length of an adjacency list is at most 3. In addition, the doubling technique to calculate the rank on the traversal list and identify the advance edge or retreat edge, need $O(\log n)$ time. Next when each processor is assigned to each element of the list, based on the informations carried by each edge whether the left branch or the right branch, an advance edge or a retreat edge, each processor can distinguish in $O(1)$ time whether each edge can satisfy the conditions from (a) to (c) or not. The condition (d) is able to be computed in $O(\log n)$ time. The condition (e) can be computed in $O(1)$ time. Furthermore, we can compute the rank of the final sublist in $O(\log n)$ time. As a result, the proposed parallel algorithm can calculate the inorder number in $O(\log n)$ time with $O(n)$ processors.

3 Parallel Algorithm for Constructing An Almost Complete Balanced Binary Search Tree

Balanced tree, like the AVL tree, improves the performance of binary search tree by keeping the tree in which for every node the heights of its two subtrees differ by at most 1 at all times. An almost complete balanced tree is a tree in which for every node both the left and right subtrees contain an equal or nearly equal amount of nodes. To describe it in more detail we consider the given binary search tree with the already determined inorder number of each node. The conversion of the given binary search tree into its almost complete balanced binary search tree is shown in Fig.6.

Fig.6(a) shows the given binary search tree with F as the root node while Fig.6(b) shows the converted almost complete balanced binary search tree. The values written on the right of each node are the inorder numbers of the given binary search tree. The values written on the left of each node are the inorder numbers of the complete balanced binary search tree added the external nodes shown by the square in Fig.6(b).

3.1 Preliminary

In what follows we give the method to number the nodes in inorder of the complete balanced binary search tree.

If N is the total number of nodes in the complete balanced binary search tree then we have $N = 2^L - 1$, here L is the maximum number of levels of the complete balanced binary search tree.

(a) If M is the inorder number of a node of jth from the left side at level i in the complete binary search tree, then it can be represented as follows.

$$M = 2^{L-i} + (j-1)2^{L-i+1} \tag{1}$$

(b) Let M be the inorder number of a node that has children at level i in the complete binary search tree, then we define M_1 and M_2 that are the inorder number of the left and the right child respectively.

Thus, we get the formula for M_1 and M_2 as follows

$$M_1 = M - 2^{L-i-1} \tag{2}$$

$$M_2 = M + 2^{L-i-1} \tag{3}$$

(c) Next task is to find the height $H(M)$ of a node with inorder number M on the complete

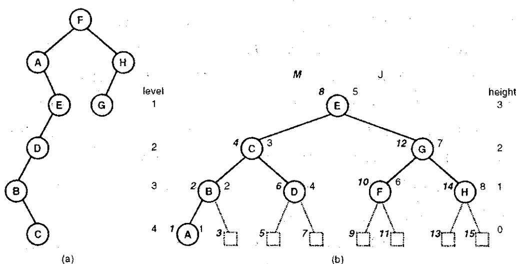


Fig. 6: Conversion of binary search tree into its complete balancing tree

balanced binary search tree. Here provided that $h = L - i$ in Eq.(1). $H(M)$ is given below where the height of the leaf node is 0.

$$h|M = 2^h * (2j-1),\ h = 0,1,\cdots,L-1;\quad j = 1,2,\cdots,2^{L-h-1}$$

$$H(M) = h \tag{4}$$

In what follows we assume $N < 2^L - 1$, that is, a binary search tree may be an almost complete balanced binary search tree. In this case, from level 1 to $L-1$, the tree is complete tree, at level L the nodes are located from left to right orderly.

(d) We define a new variable NL which denotes the number of existing nodes at level L, while another new variable S is introduced to denote the number of empty nodes at level L. NL and S can be calculated as follows.

$$NL = N - 2^{L-1} + 1 \tag{5}$$

$$S = 2^L - 1 - N \tag{6}$$

(e) If J is the inorder number in the given binary search tree, we give the following relationships for the inorder number M of each node of the complete balanced binary search tree.

$$M = \begin{cases} J & , \text{ if } J \le 2*NL \\ 2(J-NL) & , \text{ if } J > 2*NL \end{cases} \tag{7}$$

In the traditional approach the formulations from (1) to (7) have been discussed.

By the way, we already have been studied how to represent a binary search tree in form of data structure. Along with the data structure we define a new variable $LINK$ given by $LINK = array[inorderindex]$ of index, where $inorderindex$ and index denote the inorder number and index of array T respectively.

3.2 Traditional Algorithm

The traditional parallel algorithm[3] for balancing binary search tree according to the above formulations has been given in Fig.7, where in the process of conversion of the given binary search tree into its complete balanced binary search tree, first of all the inorder number of each node is calculated by Eq.(7), provided that its tree is the complete binary search tree. Further, the nodes at level L of the complete balanced binary search tree contain the odd value of inorder number M, while the nodes of levels from 1 to $L-1$, namely the internal nodes contain the even number. The internal nodes have the left and right child in according to the Eq.(2) and Eq.(3) respectively.

One of the major drawback of the traditional algorithm is that if the given binary search tree is not the complete tree, namely $N < 2^L - 1$, on the level L of the converted binary search tree, several empty nodes are attached (see Fig.6(b)), and the algorithm has assigned the processors to these empty nodes also which needlessly occupied the memory spaces. The traditional algorithm thus, has required $O(N+S)$ number of processors, where N stands for existing nodes and S stands for empty nodes of the complete balanced binary search tree.

```
procedure Para_Balance;
begin
  for each processor in parallel do begin
  { processors P1,P2,···,PN in parallel do }
    k ← T↑.num;
    if k ≤ 2*NL then
      M ← k
    else
      M ← 2*(k-NL);
    if odd(M) then begin        {leaves}
      T(LINK[k])↑.left ← nil;
      T(LINK[k])↑.right ← nil
    end
    else begin
      I ← L-H[M]; J ← 2^(L-I-1);
      T(LINK[M])↑.left ← LINK[M-J];
      T(LINK[M])↑.right ← LINK[M+J]
    end
  end
end;
```

Fig. 7: The traditional parallel balancing algorithm

Thus the data structure representation for the almost complete balanced binary search tree of the given Fig.6(a) is shown in Fig.8. In Fig.8 we can see that there are the needless memory spaces from indexes 9 to 15 of array T.

3.3 Proposed Algorithm

In this section, as shown in Fig.8 the array T and $LINK$ contain a lot of spaces for the empty nodes. Therefore, a new algorithm is proposed, which excludes the spaces for the empty nodes from the array T and $LINK$ and also reduces the number of processors assigning. Assuming the complete balanced binary search tree, inorder number M_1 (left child of M) and M_2 (right child of M) are calculated by the Eq.(2) and Eq.(3). But in an almost complete balanced binary search tree, that is, $N < 2^L - 1$, we need some modifications of those formulas. We modify Eq.(2) and Eq.(3) to get the Eq.(8) and Eq.(9) as follows, where J_1 and J_2 are the inorder number of the left and right children of the parent with inorder number J respectively.

Here, we derive the Eq.(8) and Eq.(9) to calcu-

index	key	num	left	right	Array LINK
	Array T				
1	E	8	2	3	8
2	C	4	4	5	4
3	G	12	6	7	9
4	B	2	8	9	2
5	D	6	10	11	10
6	F	10	12	13	5
7	H	14	14	15	11
8	A	1	0	0	1
9	-	3	0	0	12
10	-	5	0	0	6
11	-	7	0	0	13
12	-	9	0	0	3
13	-	11	0	0	14
14	-	13	0	0	7
15	-	15	0	0	15

Fig. 8: Data structure representation for almost complete balanced binary search tree of the traditional algorithm

late J_1 and J_2 from J as mentioned below. Note that the inorder values of J, J_1 and J_2 are the inorder number of existing nodes. The level of J in the almost complete balanced binary search tree is represented by i, which also denotes a level in the complete binary search tree.

$$J_1=\begin{cases} J-2^{L-i-1} & , J \le 2*NL \\ 2(J-NL)-2^{L-i-1} & , 2*NL < J \le 2*NL+2^{L-i-2} \\ J-2^{L-i-2} & , J > 2*NL+2^{L-i-2} \end{cases} \tag{8}$$

$$J_2=\begin{cases} J+2^{L-i-1} & , J \le 2*NL-2^{L-i-1} \\ J/2+NL+2^{L-i-2} & , 2*NL-2^{L-i-1} < J \le 2*NL \\ J+2^{L-i-2} & , J > 2*NL \end{cases} \tag{9}$$

Proof In what follows we prove the correctness of the above formulas given by Eq.(8) and Eq.(9), where NL is the number of nodes at the maximum level L as mentioned previously.

At first we argue the correctness of Eq.(8) which is divided into three possible cases depending upon the value of inorder number J of a parent node. In first case we see that if the inorder number J of parent is smaller than or equal to $2*NL$ then, from binary search tree property, the inorder number J_1 of left child also will be smaller than $2*NL$. Therefore, inorder number of left child is given as similar to Eq.(2), $J_1 = J - 2^{L-i-1}$. Mathematically, if $J \le 2*NL$ and $J_1 < 2*NL$ then from Eq.(7) we have $M = J$ and $M_1 = J_1$ respectively. Therefore from Eq.(2) the inorder number J_1 of the left child is given by $J_1 = J - 2^{L-i-1}$.

In the second case if the inorder number J of parent is greater than $2*NL$, but the inorder number J_1 of its left child is smaller than or equal to $2*NL$, mathematically, if $2*NL < J$ and $2(J-NL)-2^{L-i-1} \le 2*NL$ from Eq.(2) and Eq.(7), that is, if $2*NL < J \le 2*NL+2^{L-i-2}$, thus we get $J_1 = 2(J-NL)-2^{L-i-1}$.

In the third case if both the inorder number J of parent and the inorder number J_1 of its left child are greater than $2*NL$, namely, the conditions of $J > 2*NL$ and $2(J-NL)-2^{L-i-1} > 2*NL$ become $J > 2*NL+2^{L-i-2}$. Hence we get $J_1 = J-2^{L-i-2}$ from $2(J_1-NL) = 2(J-NL)-2^{L-i-1}$ using of Eq.(2) and Eq.(7). As a result if $J > 2*NL+2^{L-i-2}$ then $J_1 = J-2^{L-i-2}$.

Secondly we prove the correctness of Eq.(9) as similar to do Eq.(8).

In the first case that if the inorder number J_2 of the right child is smaller than or equal to $2*NL$, in this condition the inorder number J of its parent also will be smaller than $2*NL$. Deriving the result from Eq.(3) and Eq.(7) we get that if $J \le 2*NL-2^{L-i-1}$ then $J_2 = J+2^{L-i-1}$.

In the second case that if the inorder number J of parent is smaller than or equal to $2*NL$, but the inorder number J_2 of its right child is greater than $2*NL$, in this case we have $M = J$, $M_2 = M+2^{L-i-1}$ and $M_2 = 2(J_2-NL)$, thus we get $J_2 = J/2 + NL + 2^{L-i-2}$. Mathematically, if $J \le 2*NL$ and $J_2 > 2*NL$, in these conditions $J_2 > 2*NL$ which is equal to $J/2+2^{L-i-2}+NL > 2*NL$, hence this inequality becomes $J > 2*NL-2^{L-i-1}$. Therefore these conditions become $2*NL-2^{L-i-1} < J \le 2*NL$. As a result if $2*NL-2^{L-i-1} < J \le 2*NL$ then $J_2 = J/2+NL+2^{L-i-2}$.

In the third case if the inorder number J of parent is greater then $2*NL$, that is, the condition contains $J_2 > 2*NL$. As a result, from Eq.(3) and Eq.(7) we get that if $J > 2*NL$ then $J_2 = J+2^{L-i-2}$. □

From the above arguments, we have proved the correctness of both Eq.(8) and Eq.(9). Finally, we propose an efficient parallel algorithm for constructing an almost complete balanced binary search tree, which is performed in two steps as below.

1. Assign a processor to each node of the array T and the inorder traversing number of each node of the binary search tree ($T[j].num$) is sorted by array $LINK$ as $LINK[T[j].num] := j$.
2. Assign a processor to each element of array $LINK$ and by using the $LINK$ information, create an almost complete balanced binary search tree on array T in parallel.

In the proposed parallel algorithm the each size of array T and array $LINK$ is N (number of ex-

isting nodes). Therefore the proposed parallel algorithm can reduce the number of processors and the memory spaces from $O(N+S)$ to $O(N)$. Finally we propose an efficient parallel algorithm for constructing an almost complete balanced binary search tree in Fig.9. In the proposed algorithm, the computational time complexity of the proposed parallel algorithm is in $O(1)$ time with $O(n)$ processors.

4 Conclusion

Two efficient parallel algorithms have been proposed in CREW-PRAM model, one is to number the node in inorder on a binary search tree in $O(\log n)$ time and $O(n)$ processors by using the Euler tour technique, the other can construct an almost complete balanced binary search tree in constant time using the inorder number and can reduce both the number of processors and memory spaces, as compared with the traditional algorithm.

```
procedure PAR_BALANCING(T,N);
var i, j, NL, L, M: integer;
begin
  L ← ⌈log(N + 1)⌉;
   { maximum level of complete binary search tree }
  NL ← N-2^(L-1)+1;  { number of nodes on level L }
  { Phase1 : }
  for i := 1 to N in parallel do
  { Processor P_i }
     LINK[T[i].num] ← i;
  { Phase2 : }
  for j := 1 to N in parallel do
  { Processor P_j }
  begin
     if j <= 2*NL then  { calculation of indeces }
        M ← j
     else
        M ← 2*(j-NL);
     { change pointers to its children }
     if odd(M) then begin
        T[LINK[j]].left ← 0;
        T[LINK[j]].right ← 0
     end
     else if even(M) then begin
        K ← 2^(H(M)-1);
        { calculation of left child }
        if odd(M-K) and (M-K) > 2*NL then
           T[LINK[j]].left ← 0
        else if j <= 2*NL then
           T[LINK[j]].left ← LINK[j-K]
        else if j <= 2*NL+K/2 then
           T[LINK[j]].left ← LINK[2*(j-NL)-K]
        else
           T[LINK[j]].left ← LINK[j-K/2];
        { calculation of right child }
        if odd(M+K) and (M+K) > 2*NL then
           T[LINK[j]].right ← 0
        else if j < 2*NL-K then
           T[LINK[j]].right ← LINK[j+K]
        else if j <= 2*NL then
           T[LINK[j]].right ← LINK[(j+K)/2+NL]
        else
           T[LINK[j]].right ← LINK[j+K/2]
     end
  end
end.
```

Fig. 9: Proposed parallel algorithm for constructing an almost complete binary search tree

References

1) Chang, H. and Iyengar, S. S. : Efficient Algorithm to globally balance a binary search tree, *Comm. of the ACM*, Vol.27, No.7, pp.695-702 (1984).
2) Moitra, A. and Iyengar, S. S. : Derivation of a parallel algorithm for balancing binary tree, *IEEE Trans. on Software Engineering*, Vol.SE-12, No.3, pp.442-449 (1986).
3) Haq, E. and Cheng, Y. and Iyengar, S. S. : New algorithms for balancing binary search trees, *IEEE SOUTHEASTCON*, pp.378-382 (1988).
4) Gerasch, T. E. : An insertion algorithm for a minimal internal path length binary search tree, *Comm. of the ACM*, Vol.31, No.5, pp.579-585 (1988).
5) Gibbons, A. and Rytter, W. : *Efficient parallel algorithms*, Cambridge University Press, pp.21-24 (1988).
6) Migita, M. and Nakamura, R. : Parallel Algorithm for Inorder Traversal of a Binary Search Tree, *(in Japanese) IPSJ Journal*, Vol.41, No.10, pp. 2941–2944 (2000).