アルゴリズム 32－1
（1993. 3. 18）

平衡二分木を構成する並列アルゴリズム

岩下 博一，中嶋 卓雄，中村 良三
熊本大学 工学部

本論文では，EREW SM SIMD 並列計算機モデルのもとで，平衡二分木を動的に構成する実用的な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，見出し（キー）を通りがけ順になぞることによって完全木を動的に構成するアルゴリズムを基本としている．このアルゴリズムの時間計算量は，二分探索木の節点数を n，レベルを $l(l = \lfloor \log(n+1) \rfloor)$ およびプロセッサの数を N としたとき，最悪の場合 $(n = 2^{l+1} - 2)$，$O(2l + n/N)$ となり，プロセッサ数が $N \leq n/2l$ の範囲でコスト最適となる．

A Parallel Algorithm to Construct a Balancing Binary Tree

Hirokazu Iwasita, Takuo Nakashima, Ryozo Nakamura

Faculty of Engineering, Kumamoto University
2-39-1, Kurokami, Kumamoto-shi ,860 ,Japan

This paper presents a parallel insertion algorithm for constructing a complete balancing binary tree based on a sequential one by Gerasch. The algorithm is designed to run on an EREW SM SIMD parallel computer with N processors. Since its worst-case running time on a tree with n nodes and l levels is $O(2l + n/N)$. The designed parallel algorithm is cost optimal provided that $N \leq n/2l$.

1 はじめに

二分探索木はその要素（見出し）の順序関係に基づき，効率的に検索できるデータ構造である．特に，n 個の節点を持つ完全にバランス化された平衡二分木では，挿入や探索の操作が高々$O(\log n)$で可能である．したがって，平衡二分木はデータベースなど頻繁に検索操作が行なわれるシステムにおいて重要なデータ構造である．しかし，バランス化されなければ二分探索木は線形リストになることもあり，最悪の場合，時間計算量は$O(n)$となる．したがって，検索が重要な操作であるシステムにおいては，平衡な二分探索木をいかに効率良く構成するかが問題となる．

これまで，二分探索木のバランス化アルゴリズムは，なぞる節点の範囲とバランス化するタイミングとに基づき分類されてきた．

前者のなぞる節点の範囲に注目すると，バランス化アルゴリズムは二分探索木の節点を部分的になぞるローカルバランス化アルゴリズムとすべての節点をなぞるグローバルバランス化アルゴリズムとに分類される [1].

後者のバランス化するタイミングに注目すると，二分探索木に対する操作毎にバランス化を行なう動的バランス化アルゴリズム [5] と周期的にバランス化を行なうアルゴリズム [7][8] に分類できる [4].

バランス化アルゴリズムはなぞる節点の範囲とタイミングの組合せによって，種々のアルゴリズムが提案されている．たとえば，AVL アルゴリズムは，キーを挿入する時点でバランス化条件を判定し，もしバランス化が必要なら，部分木を回転させ，バランスを保持する動的なローカルバランス化アルゴリズムである．一方，Chang らによって提案されたアルゴリズムは [2][3]，二分探索木全体の再構成のため，木のすべての節点を調べ，完全に (*completely*) バランス化した平衡二分木を構成する周期的なグローバルバランス化アルゴリズムである．

Gerasch は，このローカルバランス化とグローバルバランス化アルゴリズムの長所を兼ね備えた動的なアルゴリズムを提案している [1]. Gerasch のアルゴリズムでは，ローカルバランス化の長所である部分的な節点のなぞりによって，グローバルバランス化の長所である完全にバランス化された平衡二分木を構成することができる．このアルゴリズムは最悪の場合には，すべての節点を探索するグローバルバランス化アルゴリズムとなる．

一般に，動的バランス化アルゴリズムは挿入・探索の操作ごとに動作するため，負荷が軽いローカルバランス化アルゴリズムであり，周期的バランス化アルゴリズムは動作する頻度が少ないため，グローバルバランス化アルゴリズムである．

従来，平衡二分木を構成する並列アルゴリズムは，周期的に二分探索木を平衡化するグローバルバランス化アルゴリズムについて考察されているが [4][7][8]，しかし，動的な並列バランス化アルゴリズムについては，ほとんど考察されていない．

本稿では，見出しを通りがけ順になぞることによって完全木を動的に構成するアルゴリズムを基本とし，EREW SM SIMD 計算機モデルのもとで，任意の個数のプロセッサで動作し平衡二分木を動的に構成する並列アルゴリズムを提案する．

まず，2 章では，Gerasch による平衡二分木を構成する逐次アルゴリズムを概観し，その時間計算量を解析する．3 章では，任意個のプロセッサで動作する動的な並列アルゴリズムを提案し，4 章では，提案した並列アルゴリズムの時間計算量を解析し，評価する．

2 従来の平衡二分木を構成するアルゴリズム

本稿で用いる用語は Knuth の文献 [6] の定義に従うものとし，次の二つを新たに定義する．

定義 1 (レベル l の準完全木) 二分探索木のレベル $l-1$ までが完全木で，レベル l はひとつも節点を持たないか，あるいは *1* 個以上の空節を持つ．このレベル l を最大レベルと呼ぶ．このとき，二分探索木はレベル l の準完全木 *(nearly complete tree)* であるという．ここでは，根のレベルは *0* とする．

定義 2 (完全レベル) 端節 *(terminal node)* の完全レベル *(complete level)* を 1 とする．枝節 *(branch node)* の完全レベルは，その左右の部分木の根節点における完全レベルの小さい方に 1 を加えた値である．

上記の定義により，節点の完全レベルとは，その節点を根節点とした部分木が何レベルの準完全木になっているかを示す値となる．

たとえば，図1にレベル3の準完全木を示す．さらに，図の各節点の右に，その節点の完全レベルを付記している．

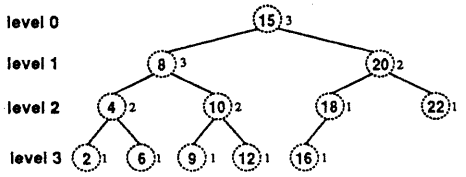


図1 レベル3の準完全木の例

2.1 逐次アルゴリズム

Gerasch[1]によって提案された逐次バランス化アルゴリズムでは，各節点は，キー，左右の部分木へのポインタおよび完全レベルを保持している．また，アルゴリズムは再帰手続きで構成され，後戻りを利用して通りがけ順(*inorder*)に探索しながらキーを移し替え平衡木を維持している．

レベルlの準完全木の最大レベルlにある節点にキーが挿入される場合，次のようにバランス化アルゴリズムが動作する．

まず，挿入されるキーは，各節点の完全レベルを参照しながら挿入される．ある節点において，次に挿入される側の部分木の完全レベルが，他方の部分木の完全レベルより大きいとき，その節点より下の部分木において不平衡が起きるので，その節点をpivot節点と呼ぶ．

その後，挿入は不平衡の状態を保持したまま最大レベルlまでたどり着き，この時点からキーの移し替えが始まる．

この過程で，pivot節点から挿入が行なわれた左部分木中（右部分木）の最大（最小）のキーがpivot節点のキーとなり，いままでのpivot節点のキーは他方の部分木へ新しいキーとして挿入される．以降は前述した挿入と同様の動作を繰り返す．

その結果，新しい節点の挿入によって，平衡二分木を維持できるようにキーが通りがけ順に移し替えられる．そして，挿入された節点からpivot節点までのパス上の各節点の完全レベルが変更される．

たとえば，図1に示す二分探索木にキー5を挿入したとき，節点⑮においてその子節点である節点⑥と節点⑳の完全レベルがそれぞれ3と2であるので，節点⑮がpivot節点となる．次に，キー5が節点⑮の左部分木に挿入され，さらに節点⑧の左部分木，節点④の右部分木と挿入され節点⑥にたどり着き，節点⑥にキー5が挿入さる．この時点から，後戻りしながら通りがけ順に節点のキーの移し替えを始める．pivot節点までくると，キー12がpivot節点のキーの値となる．pivot節点の元のキー15は右の部分木への新しい挿入キーとなる．このような挿入と移し替えが最終的に終了した時点の二分探索木を図2に示す．ここで，キーの右上に付した*（アスタリスク）は，バランス化の過程でキーが変更された節点を示す．

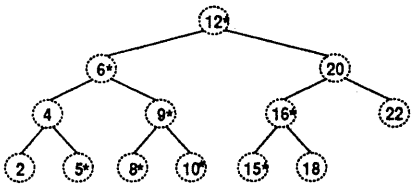


図2 挿入が終了した時点の二分探索木

2.2 最悪の場合の解析

Geraschの逐次アルゴリズムの時間計算量は節点をなぞる毎に挿入手続きを再帰的に呼び出すので，実行時間は，通りがけ順になぞる二分探索木の節点の個数に依存する．

一般に，二分探索木の最大レベルに配置されている節点のパターンの多様さから，平均的な時間計算量の評価は複雑である．ここでは，最悪，すなわち最大レベルのすべてのキーが移し替えの対象となる場合について考察する．

まず，レベルlの準完全木において最大レベルlの節点の個数をkとしたとき，二分探索木の節点の個数nは，次のように表される．

$$n = 2^l - 1 + k \quad (1)$$

$$l = \lfloor \log_2(n+1) \rfloor \quad (\text{ただし } 0 \leq k < 2^l) \quad (2)$$

最悪の場合とは，最大レベルlにk個の節点が最

左（最右）から連続的に配置されている二分探索木に，最小（最大）のキーを挿入する場合である．

たとえば，図3に示す準完全木にキー1を挿入する場合が最悪の場合である．

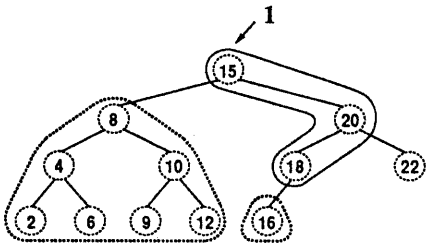


図3 最悪の場合における準完全木

図3では，逐次アルゴリズムが通りがけ順になぞる節点を複数の線で囲んでおり，これらの線で囲まれた節点の総数が時間計算量に対応すると考えることができる．

図3の線で囲まれた節点は二つのグループに分類される．ひとつのグループは，根から挿入される節点までの経路上にある節点で図では実線を用いて囲んでいる．他のグループは，最大レベルの節点を端節とする完全木を構成する節点で点線を用いて囲んでいる．それらグループ内の節点の個数は次のように計算できる．

まず，根から経路上の節点の数は，木のレベル l で表される．

次に，複数の完全木を構成する節点の個数は，最大レベル l における節点の個数 k から計算できる．まず，最大レベルの節点の個数 k を2進数で表現したとき，'1' の個数を表す関数 $t(k)$ を導入する．

たとえば，この関数 $t(k)$ の値は k が1から6まで変化するとき，次のような値となる．

kの値	2進数	$t(k)$
1	1	1
2	10	1
3	11	2
4	100	1
5	101	2
6	110	2

k を2進数で表現すると，'1' の個数が完全木の個数を表し，その桁が各完全木の高さを表すことができる．たとえば図3において，$k=5$ であるから2進数で表すと101となり，高さ3と高さ1の完全木から構成されているのがわかる．

これにより，完全木を構成する節点の総数は $2k-t(k)$ と表すことができる．

したがって，最悪の場合，なぞられる節点の個数 $W(n)$ は，次のようになる．

$$W(n) = l + 2k - t(k) \tag{3}$$

ただし　k ： 木の最大レベルの節点数

$t(k)$ ： k を2進数で表したとき'1'の個数を返す関数

したがって，逐次アルゴリズムの最悪な場合の時間計算量 $t_s(n)$ は，

$$t_s(n) = l + 2k - t(k) \tag{4}$$

$$= O(l + 2k) \tag{5}$$

となる．

この挿入アルゴリズムにおいて，最大レベル l の節点の個数 k が $0 \leq k < 2^l$ の範囲にあることから，$O(\log n)$ と $O(n)$ の間の計算量となる．したがって，k が最大となる $k = 2^l - 1$ のとき，すなわち $n = 2^{l+1} - 2$ の場合には，

$$t_s(n) = O(n) \tag{6}$$

となる．

また，平衡二分木を構成する逐次バランス化アルゴリズムにおいては，$O(n)$ より少ないステップ数になり得ないので，逐次バランス化アルゴリズムの下界は $\Omega(n)$ である．

3 並列アルゴリズム

本章では，Gerasch の提案する逐次アルゴリズムに基づき，動的な並列挿入・バランス化アルゴリズムを提案する．まず，提案する並列アルゴリズムの仮定を次のように定める．

アルゴリズムの仮定

1. 提案する並列アルゴリズムは $P_1, P_2, \cdots, P_N$ の N 個のプロセッサを持つ EREW SM SIMD 並列計算機モデル上で動作する．
2. バランス化された二分探索木の節点を，配列 Tree の要素として共有メモリ (SM) に格納する．

二分探索木のデータ構造

二分探索木のデータ構造には，次のような配列 Tree を用いる．

1. 二分探索木の根は Tree[1] である．
2. 節点 Tree[i] の左の子は Tree[$2i$]，右の子は Tree[$2i+1$] である．
3. 各節点 Tree[i] は完全レベルの値を保持する．

図4は図1の二分探索木を配列 Tree で表したものであり，配列 Tree の各要素はキーの値と完全レベルを保持する．

	1	2	3	4	5	6	7	8	9	10	11	12
TREE	15	8	20	4	10	18	22	2	6	9	12	16
	3	3	2	2	2	1	1	1	1	1	1	1

図 4 二分探索木のデータ構造

3.1 並列アルゴリズム

Gerasch の逐次アルゴリズムは，キーを挿入するアルゴリズムと木を通りがけ順になぞりながらキーを移し替えるバランス化アルゴリズムから構成されている．

そこで，提案する並列アルゴリズムは次のようなステップから構成される．

step1：挿入されたキーが配置される位置と移し替えによって新しい節点が配置される位置の探索．

step2：通りがけ順（または通りがけ順の逆順）のキーの移動．

step3：完全レベルの変更．

以下では，各ステップ毎にアルゴリズムを詳細に説明する．

Step1

プロセッサ P_1 は二分探索木の根から順に各節点のキーと挿入キーとを比較し，挿入キーが配置される配列 Tree 上の位置を求める．さらに各節点でその子の完全レベルを比較することによって，この挿入が不平衡を起こすかどうかを調べる．

もし不平衡が起こらず最大レベルに到達すれば，挿入される位置は必ず空節であるため，挿入キーをその節点に登録して，アルゴリズムは終了する．

一方，不平衡が起きれば，その節点すなわち pivot 節点において，プロセッサ P_1 の他に新たなプロセッサ P_2 が起動され，並列に動作する．P_1 は挿入キーが配置される場所を探し続ける．一方，P_2 は P_1 とは別の部分木に pivot 節点のキーを挿入キーとして，P_1 と同じ挿入動作をする．一度不平衡が生じた場合，P_1 による挿入操作では新たな不平衡は起こらず，最大レベルにおいて挿入位置 i を求める．P_2 による挿入では不平衡が起きる可能性がある．不平衡が起きたときには，前述の P_2 が行なった動作と同様な動作を繰り返す．その結果，最大レベルの新たに節点が配置される位置 j を求める．

この P_1 と P_2 による挿入によって最終的にそれぞれ最大レベルの端節 Tree[i] と端節 Tree[j] に到達する．すなわち，キーの移し替えを始める節点 Tree[i] と終了する節点 Tree[j] の位置が得られる．(step1 の終了)

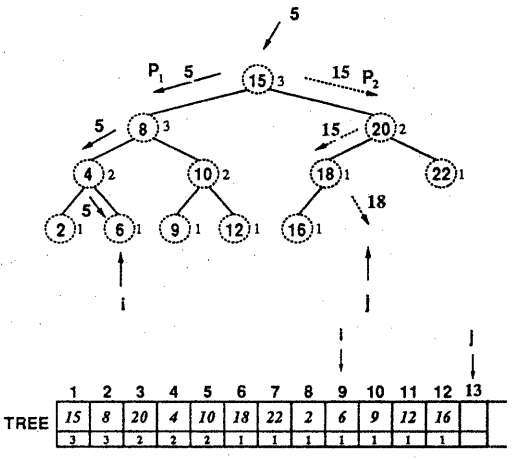


図 5 Step1 が終了した時点の二分探索木とそのデータ構造

たとえば，図1に示した二分探索木にキー5を挿入したときの P_1 と P_2 の動作について説明する．まず，P_1 が挿入キー5を根節点⑮と比較する．この時に根節点の2つの子の完全レベルは異なるため不平衡が起きる．P_1 は5を左部分木に挿入し，並列に P_2 は15を右部分木へ挿入する．プロセッサ P_1

は節点⑧，節点④をだどり，最終的に節点⑥に到達し，その配列 Tree 上の位置 9 を得る．そのときの動作の流れを図5では矢印をつけた実線で示している．一方，P_2は，節点⑳，節点⑱をだどり，最終的に 18 を挿入キーとして，空の節点である配列上の位置 13 を得る．そのときの動作の流れを図5では破線の矢印で示している．

Step2

$i < j$の場合には pivot 節点の左部分木の節点 Tree[i] から右部分木の節点 Tree[j] への右方向，すなわち通りがけ順にキーが移し替えられ，$j < i$の場合には左方向，すなわち通りがけの逆順にキーが移し替えられる．以下では，それぞれの場合について処理を説明する．

$i < j$の場合

位置 i から位置 j までの通りがけ順にある範囲の節点を，次のように N個のプロセッサで均等に分割する．

Tree[i],Tree[$i+1$],$\cdots$,Tree[j] の節点はすべて端節となり，通りがけ順に節点を並べた時その端節は 1 つおきに配置される．したがって，i から j までの連続した $j-i$ 個の端節を等間隔 $h = \lceil \frac{j-i}{N} \rceil$ に分割し，N個の各プロセッサにそれぞれ割り当てる．このことは，移動するすべての節点を N個のプロセッサに等分することになる．

このアルゴリズムではキーを最終的に移動する節点を考慮して，各プロセッサに端節を $h+1$ 個割り当てる．したがって，各プロセッサ $P_k(1 \leq k \leq N)$ はそれぞれ Tree[$i+h(k-1)$] から Tree[$i+hk$] までの移し替えを並列に行う．

各プロセッサはそれぞれ割り当てられた節点に対して以下の手順でキーを移す．

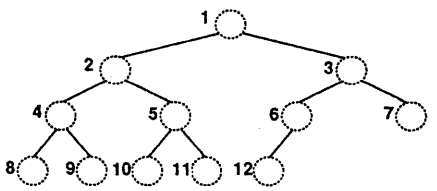


図 6 二分探索木における節点番号

二分探索木に図6に示すような番号を幅優先に左から右に付ければ，通りがけ順になぞるとき，節点番号 i の次になぞられる節点番号 $next(i)$ は次のように表せる．

$$next(i) = [i = even \rightarrow next := i \,\mathrm{div}\, 2, next(i \,\mathrm{div}\, 2)] \tag{7}$$

したがって，端節 Tree[i] のキーを移動する節点番号 kは $k = next(i)$ となり，その節点は Tree[k] となる．

求められた端節の移動先に対して，端節 Tree[i] のキーを節点 Tree[k] へ移動し，元の節点 Tree[k] のキーを端節 Tree[$i+1$] に移動する．

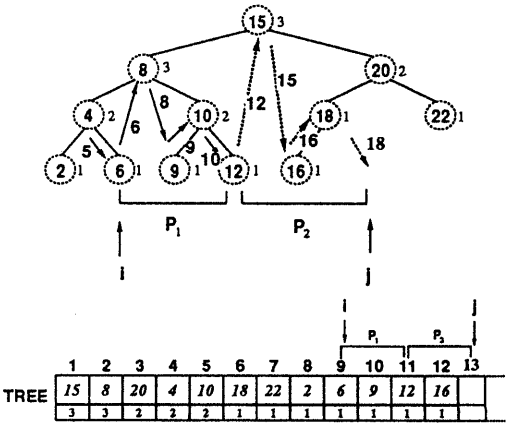


図 7 Step2 において，右方向に節点を移動する場合の例

たとえば，図 5の例において，プロセッサ数が $N = 2$ の場合を図 7に示す．

この例では，$i = 9$, $j = 13$ となるので，端節の個数は $13-9 = 4$ 個となる．ここで，2 個のプロセッサで均等に分割したとき，その間隔は $h = \lceil \frac{13-9}{2} \rceil = 2$ となる．したがって，プロセッサ P_1が端節 Tree[9] から端節 Tree[9+2]=Tree[11] までの移し替えを行ない，プロセッサ P_2が Tree[11] から Tree[13] までの移し替えを行なう．

端節 Tree[9] のキー 6 の移動先は Tree[2] となる．また，Tree[2] のキー 8 の移動先は Tree[10] となる．

このように，二分探索木を配列で表現することにより，簡単に通りがけ順の移動を実現することができる．

$j < i$の場合

位置jから位置iまでの範囲にあるすべての節点を，$i < j$の場合と同様にN個のプロセッサで均等に分割する．

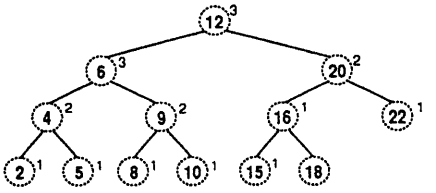
二分探索木の節点に番号付け，通りがけの逆順になぞるとき，節点番号iの次になぞられる節点番号$next(i)$は次のように表せる．

$$next(i) = [i = odd \rightarrow next := i \,\mathrm{div}\, 2, next(i \,\mathrm{div}\, 2)] \tag{8}$$

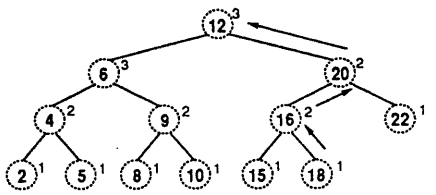
したがって，端節Tree[i]のキーを移動する節点番号kは$k = next(i)$となる．その節点はTree[k]となる．

求められた端節の移動先に対して，端節Tree[i]のキーを節点Tree[k]へ移動し，元の節点Tree[k]のキーを端節Tree[$i-1$]へ移動する．

このようにして位置iから位置jまでの範囲にあるすべての節点を移動する．(step2終了)



(a) 完全レベル変更前の平衡木



(b) 完全レベル変更後の平衡木

図8 Step3における完全レベルの変更

Step3

step1の挿入時に決定した，根から新しい節点が配置される節点Tree[j]までの経路上の節点についてのみ完全レベルを変更する．すなわち，節点Tree[j]から順にその親節点について木の根まで完全レベルを変更する．(step3終了)

たとえば，図8(a)にStep2が終了したときの平衡木を示す．新たに生成された節点⑱は配列Treeのインデックス$j = 13$に位置する．節点⑱は端節であるので，完全レベルは1となり，その親Tree[$\lceil \frac{13}{2} \rceil$]=Tree[6]である節点⑯は，その左右の部分木の根であるTree[12],Tree[13]の完全レベルが1であることから，完全レベルを2に変更する．同様にして，順次親をたどり，木の根節点まで完全レベルを変更する．

4 解析

提案する並列アルゴリズムを各ステップ毎に解析する．

まず，step1はプロセッサP_1およびP_2による根節点から端節までの経路上の節点との比較であるため，木のレベル$l(l = \lfloor \log(n+1) \rfloor)$に比例した$O(l)$の実行時間がかかる．

step2では，Tree[i]からTree[j]までの端節に等間隔に割り当てられたN個のプロセッサが並列にキーの移し替えを行う．ここでTree[i]のキーからTree[j]のキーまでの範囲にある節点の個数は$2|i-j|$となり，各プロセッサは$2|i-j|/N$個の節点に対しキーの移し替えを行うので，$O(2|i-j|/N)$の実行時間がかかる．

step3では根からTree[j]までの経路上の節点について完全レベルが変更されるので，$O(l)$の実行時間がかかる．

したがって，全実行時間は次のようになる．

$$t(n) = O(l) + O(2|i-j|/N) + O(l) \tag{9}$$

ここで，節点数が$n = 2^{l+1} - 2$すなわち$2|i-j| = n$のとき，同じ最大レベルを持つ木に対して最悪となり，時間計算量$t(n)$は次のようになる．

$$t(n) = O(2l + n/N) \tag{10}$$

ここで，Step2におけるプロセッサ数Nは，キーの移し替えを開始する位置iと終了する位置jまでの連続した位置に存在する端節の個数から，$1 \leq N \leq j - i$であれば，自由に設定することができる．

したがって，この並列アルゴリズムのコスト $c(n)$ は実行時間 $t(n)$ とプロセッサの数 $p(n)$ の積より、次のようになる．

$$\begin{aligned} c(n) &= t(n) \times p(n) \\ &= O(2Nl+n) \end{aligned} \tag{11}$$

したがって，この並列アルゴリズムのコスト $c(n)$ は $N \leq n/2l$ のとき，逐次バランス化アルゴリズムの下界 $\Omega(n)$ に等しくなる．したがって，この条件のもとで提案する並列アルゴリズムはコスト最適である．

5 おわりに

本論文では，EREW SM SIMD 並列計算機で動作する並列挿入・バランス化アルゴリズムを提案した．

提案した並列アルゴリズムは，配列を木で表したときの特徴を利用し，各プロセッサに均等に節点を割り当て，さらにバランス化するときのキーの移動先を決定する計算を容易にしている．また提示したアルゴリズムはプロセッサ数 N が $N \leq n/2l$ のとき，コスト最適であることを示した．

時間計算量の解析では，最悪な場合のみの解析を行なったが，現実の現象を忠実に評価するためには，逐次および並列アルゴリズムにおいて平均的な計算量も解析する必要がある．

参考文献

[1] T.E.Gerasch, "An insertion algorithm for a minimal internal path length binary search tree" *Commun. of the ACM*, Vol.31, No.5, pp.579–585, 1988.

[2] Hsi Chang and S.Sitharama Iyengar, "Efficient algorithms to globally balance a binary search trees" *Commun. of the ACM*, Vol. 27, No. 7, pp. 695–702, July 1984.

[3] J.L.Bentley, "Multidimensional Binary Search Trees Used for Associative Searching" *Commun. of the ACM*, Vol. 18, No. 9,pp.509–517,1975.

[4] E.Haq, Y.Cheng and S.S.Iyengar, "New Algorithms for Balancing Binary Search Trees" *IEEE SOUTHEASTCON*, pp.378–382, 698, 1988.

[5] D.E.Knuth, "The Art of Computer Programming", Vol.3,*sorting and searching*, Addison-Weslay,1973.

[6] D.E.Knuth, "The Art of Computer Programming", Vol.1, pp.305–422, *Fundamental Algorithms*, Addison-Weslay, 1973?

[7] A.Moitra and S.S.Iyengar, "Derivation of a Parallel Algorithm for Balancing Binary Trees" IEEE Trans. on Software Engineering, vol SE–12, No. 3, pp. 442–449, March 1986.

[8] S.S.Iyengar and H.Chang, "Efficient Algorithms to create and maintain Balanced and Threaded Binary Search Trees", *Software–Practice and Experience*, Vol.15, No.10, pp.925–941, 1985.