Parallel Update and Search in Skip Lists

J. Gabarró
C. Martínez
X. Messeguer

Report LSI–93–11–R

UPC
Facultat d’informàtica de Barcelona - Biblioteca
19 JUN. 1995
Parallel Update and Search in Skip Lists

Joaquim Gabarró‡  Conrado Martínez‡  Xavier Meseguer‡

Abstract

We present a PRAM implementation of a parallel dictionary with skip lists. More precisely, we give algorithms to search, insert and delete \( k \) ordered elements in a skip list of \( n \) elements in parallel. The algorithms are clear and easy to explain and implement. All of them are iterative. They can be implemented in the EREW PRAM model using \( O(k) \) processors in expected time \( O(\log n + \log k) \).

Keywords: SIMD machines, parallel algorithms, EREW PRAM model, skip lists, dictionaries.

1 Introduction

An skip list is a randomized data structure that can be used for representing abstract data types such as dictionaries, symbol tables and ordered lists. Skip lists were introduced by Pugh in 1990 [6] and are an alternative to balanced trees; although they have bad worst-case performance, the randomization process involved in their construction guarantees an expected sequential performance of the same order of magnitude of that of balanced trees.

Sequential skip list algorithms are very simple to implement, providing a significant constant factor improvement over balanced and self-adjusting trees. On the other hand, skip lists are also space efficient, requiring an average of 2 (or less) pointers per node and no balance, priority or weight information.

Moreover, they have a rich and interesting probabilistic analysis; consider, for instance, the work done by T. Papadakis, J. I. Munro, and P.V. Poblete in [4], and by P. Kirschenhofer and H. Prodinger in [3].

In [5] W. Paul, U. Vishkin and H. Wagener developed and analyzed the algorithms to dynamically maintain a parallel dictionary on 2–3 trees. They considered the EREW PRAM model. Search, insertion and deletion algorithms for \( k \) items in a 2–3 tree of \( n \) nodes were shown to take time \( O(\log n + \log k) \) with \( O(k) \) processors in the worst-case. However, the algorithms corresponding to the insertion and deletion are quite sophisticated and use recursion. This is a handicap because most of the actual parallel programming languages (like C*) do not allow recursion.

Here, we present another implementation of a parallel dictionary with skip lists. Skip lists behave in some aspects like balanced trees and in some other aspects as linked lists. Therefore, we profit of this dual view to develop clear and easy algorithms.

‡This research was supported by the ESPRIT BRA Program of the EC under contract no. 7141, project ALCOM II.

‡Department de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, Pau Gargallo 5, 08028-Barcelona. Contact e-mail: conrado@lsi.upc.es
In section 3 we present the algorithm to search \( k \) keys in parallel. To derive it we use the skip list as a tree. The algorithm routes a set of packets containing the keys along the skip list. When a packet collides with a key it is split into two packets. In section 4 we consider insertion and deletion. We give an algorithm for insertion and sketch that for deletion. Both algorithms treat the skip list as a set of linked lists. They can be seen as a parallelization of the usual sequential algorithms for lists with some extra memory (on an array) to do parallel address arithmetic. All three algorithms are iterative.

Special care is taken to obtain clear and complete algorithms. We apply stepwise development techniques along the lines suggested in [1]. This is fundamental to be able to implement these algorithms on real machines. From a theoretical point of view, we obtain also interesting results because the expected performance of our algorithms is comparable to that given in [5].

Theorem 1.1. The search, insertion and deletion algorithms can be implemented in the EREW PRAM model with expected time \( O(\log n + \log k) \) with \( O(k) \) processors, where \( k \) is the number of given keys and \( n \) is the number of items in the skip list.

We think these algorithms are important because they fulfill (at the same time) two constraints:

1. From a programming point of view, they are simple and it is easy to reason about them. Moreover they seem to have practical implementations in real machines.

2. From a theoretical point of view, they have acceptable complexity bounds.

2 Skip lists

Let us recall some basic facts about skip lists. We shall assume that the items to be stored in a skip list \( S \) are drawn from some totally ordered set and are different. We will sometimes use the word key instead of item.

A non-empty skip list is a collection of sorted linked lists. Each list is given a level, numbered 1, 2, ..., All the \( n \geq 1 \) items of a skip list are stored in the first level list, in sorted order. Furthermore, if \( x \) is an item included in the list of level \( l \), then it is also included in the list of level \( (l + 1) \) with probability \( p \). An item is said to be of level \( l \) if it is included in exactly \( l \) lists. To implement the skip list, we need to allocate a node for each item in the skip list. Each node contains the item and as many pointers as the number of levels in the list the item was included. These pointers, called forward pointers, point to the successors of the node in each list. We also allocate a header node with pointers to the first item in each list (see Figure 2.1). We shall write level(\( S \)) to denote the maximum level of the skip list and head(\( S \)) to denote the header node of \( S \). The maximum level of a skip list \( S \) is also called height of the skip list. Given a node \( x \) of a skip list, we use level(\( x \)) and key(\( x \)) for the level and the key of the node \( x \), respectively. If \( 0 \leq l \leq \text{level}(x) \), forward(\( x, l \)) denotes the successor at level \( l \) of the node \( x \).

Moreover, we use the following conventions: 1) each node in a skip list \( S \) has a fictitious level 0; for all non-nil nodes \( x \) in \( S \), forward(\( x, 0 \)) = forward(\( x, 1 \)); 2) there is a node called NIL pointed by the last item of each of the linked lists. By convention, level(NIL) = level(\( S \)) + 1 = level(header(\( S \))) + 1; 3) the header node stores a dummy key that is less than any legal key: key(header(\( S \))) = -\( \infty \). NIL is given a key greater than any legal key: key(NIL) = \( \infty \); 4) each node stores as many copies of its key as its level indicates; this convention is needed to avoid read conflicts.
Let $x$ be a node of the skip list $S$ and let $l$ some positive integer such that $l \leq \text{level}(x)$. The first node of level $> l$ whose key is greater than key($x$) is called wall($x$, $l$). Note that for any node $x$, wall($x$, 0) = forward($x$, 1). We define the subskiplist of $S$ starting at ($x$, $l$), denoted by $S(x, l)$, as the skip list consisting of all the nodes of $S$ that lie between the node $x$ and the node wall($x$, $l$), the former excluded. Finally, let us define the following predicate: \( \text{in}_S(a, x, l) \equiv (a \in S(x, l) \lor a \not\in S) \).

**Lemma 2.1.** Let $a$ be a legal key such that for some node $x$ and a level $l > 0$, it holds in$_S(a, x, l)$. Let $b$ be the key stored at the successor of $x$ at level $l$, that is, $b = \text{key}(\text{forward}(x, l))$. Then, if $a > b$, it follows that in$_S(a, \text{forward}(x, l), l)$; otherwise, $a \leq b$ and in$_S(a, x, l - 1)$.

If $l = 0$, the truth of $a \in S(x, l)$ is easily checked: $a \in S(x, 0) \equiv (a = \text{key}(\text{forward}(x, 1)))$. Note that the sequential search algorithm [6] for skip lists just iterates while maintaining the invariant in$_S(a, x, l)$, where $a$ is the key to be searched.

### 3 Search

Given an skip list $S$ of $n$ items and an ordered array $a$ with $k$ keys, the search algorithm that we derive in this section returns an array of $k$ booleans $\text{belongs}$, such that $\text{belongs}[i] = true$ if and only if $a[i]$ is present in $S$. It runs in an EREW PRAM using $O(k)$ processors and expected time $O(\log n + \log k)$.

Fundamental to our search algorithm is the notion of *packet* (this concept is quite similar to that of *chain* that appears in [5]). A packet $p$ is an object consisting of a segment of the array $a[1..k]$ of keys to be searched, say $a[i], a[i+1], \ldots, a[j]$, for some $1 \leq i \leq j \leq k$, a reference to some non-nil node $x$ of the skip list $S$, denoted by node($p$), a level $l$, denoted by level($p$), such that $0 \leq l \leq \text{level}(x)$ and an active or inactive status, active($p$). We denote first($p$) = $i$ and last($p$) = $j$. More formally,

\[
p = <\text{first}(p) \ldots \text{last}(p)>, \text{node}(p), \text{level}(p), \text{active}(p)>
\]

The search algorithm proceeds in stages. In any given stage, the set of packets $P$ gives us a full partition of the array $a[1..k]$, since every key is hold by one and only one packet. If a packet $p$ contains a reference to node $x$ and its level is $l$, at some stage $t$, we shall say that $p$ points to or is at ($x$, $l$) during stage $t$. A given packet $p$ is active ($\text{active}(p) = true$) if and only if its level is not null, i.e. $\text{active}(p) \equiv (\text{level}(p) > 0)$. The subset of active packets in $P$ is denoted $\text{active}(P)$. A processor must be associated to each active packet to perform the movement or split of each packet in parallel (see Algorithm 1).
procedure search(in a : array [1..k] of key.type; in S : skip.list; out belongs : array [1..k] of boolean)
procedure route(in p : packet; in/out P : set.of.packets; in S : skip.list)
    var b : key.type;
    begin
        b := key(forward(node(p), level(p)));
        if
            b < a[first(p)] —— push.forward(p, P, S)
            || b ≥ a[last(p)] —— push.down(p, P, S)
            || a[first(p)] ≤ b < a[last(p)] —— split.and.push (p, P, S)
        end route;
    end
procedure split.and.push (in p : packet ; in/out P : set.of.packets; in S : skip.list)
    var b : key.type; m : [1..k];
    begin
        b := key(forward(node(p), level(p)));
        m := (first(p) + last(p)) div 2;
        split.packet(p, m, p1, p2, P, S);
        if
            b < a[first(p2)] —— push.forward (p2, P, S)
            || a[last(p1)] ≤ b —— push.down(p1, P, S)
        end split.and.push;
    end

var p : packet; P : set.of.packets;
begin
    p := <<[1..k], header(S), level(S), true>>; P := { p };
    do active(P) ——
        for all p : p ∈ active(P) do in parallel route(p, P, S) end
    od;
    for all p : p ∈ P do in parallel
        for all i : [first(p)..last(p)] do in parallel
            belongs[i] := (a[i] = key(forward(node(p), 1)))
        end
    end
end search

Algorithm 1: Search algorithm.

At the very start of the algorithm, before stage 1, an active packet containing all k keys is
"injected" into the skip list S by making it point to the header node of S and its maximum
level. In each stage, all active packets are routed through the skip list S. The procedure
route(p,P,S) moves or splits packet p in order to maintain the following invariant: for each key
a within p, it holds in.S(a, node(p), level(p)). The search algorithm finishes when active(P)
becomes empty.

Let us consider an active packet p that points (x, l) at stage t. Let b be the key of the
successor of node x at level l, that is, b = key(forward(x, l)). The procedure route pushes
forward, down or splits the packet p using the following rules:

1. If the first key in p is greater than b, then all the keys in p must be in the substkiplist
starting at level l of the successor of x of level l, or not be at all. More specifically,
in\_S(a, forward(x, l), l) holds for any key a in p, by Lemma 2.1. In that case, the packet must be pushed forward, replacing the node it points to by forward(x, l).

2. If the last key in p is not greater than b (b \geq a[\text{last}(p)]) then in\_S(a, x, l - 1) is true for all keys a in p, and the packet must be pushed down, decrementing its level. If the level of p becomes 0, then the packet is made inactive.

3. Finally, b could fall in the midst of the first and last key of the packet. We say that p hits (x, l) if and only if a[\text{first}(p)] \leq b < a[\text{last}(p)]. In that case, route calls procedure split\_and\_push. The packet p = \langle [i..j], x, l, true \rangle is halved into two packets p_1 = \langle [i..m], x, l, true \rangle and p_2 = \langle [m+1..j], x, l, true \rangle, with m = (i + j) \mod 2. Only one of p_1 and p_2 hits (x, l) and remains at the same node and level, whereas the other can be pushed forward or down applying one of the rules 1 and 2 above.

3.1 Efficiency of the search algorithm

The main theorem of this subsection gives the performance of the search algorithm.

Theorem 3.1. The search algorithm can be implemented in an EREW PRAM model with expected time O(log n + log k) and using O(k) processors, where k is the number of keys to be searched and n is the length of the skip list.

Recall that we have written the search algorithm as if we were using a CREW model. The first part of our analysis shows that there is at most a constant number of read conflicts for any given stage t and node/level. In the second part, we give a bound on the expected number of iterations or stages that have to be performed in order to search the k keys.

First, the following Lemma can be proved.

Lemma 3.1. The route procedure route(p, P, S) can be implemented in the EREW model in time O(1) with O(k) processors.

Read conflicts will appear when several packets are at some node/level (x, l). Therefore we need to consider the evolution of packets in S. Remark that all the subpackets crossing (x, l) follow a unique labeled path called zig\_zag(x, l) = [(header(S), level(S)) \xrightarrow{b_1} \ldots \xrightarrow{b_i} (x, l)]. When b_i \leq key(x) the path “goes through” a forward pointer and is said to be a forward key, otherwise b_i > key(x), the path “steps down one level” on the same node and b_i is said to be down key. The keys in the zig\_zag path to any node/level can be partitioned into forward keys f_1, f_2, \ldots, key(x) and down keys d_1, d_2, \ldots, down\_key(x, l) with f_1 < \cdots < key(x) < down\_key(x, l) \leq \cdots \leq d_1. We write

zig\_zag(x, l) = [f_1, \ldots, key(x), down\_key(x, l), \ldots, d_1].

Consider the skip list given in Figure 2.1 (for short we identify key(x) with x). The zig\_zag path to the node/level (44, 1) is

[(header, 5) \xrightarrow{29} (29, 5) \xrightarrow{\infty} (29, 4) \xrightarrow{\infty} (29, 3) \xrightarrow{46} (29, 2) \xrightarrow{41} (41, 2) \xrightarrow{46} (41, 1) \xrightarrow{44} (44, 1)].

We write zig\_zag(44, 1) = [29, 41, 46, 46, \infty, \infty]. The forward keys are 29, 41, the down keys are \infty, \infty, 46, 46 and down\_key(44, 1) = 46.

We show that no packet generates a lot of subpackets crossing (x, l) together.
Lemma 3.2. Let \( p \) be a packet in some node/level on the zig_zag\((x, l)\) path.

- when \( a[\text{first}(p)\ldots\text{last}(p)] \subseteq (\text{key}(x)\ldots\text{down\textunderscore key}(x, l)) \) the packet \( p \) never splits and will move through \((x, l)\) with no delay executing push\textunderscore forward and push\textunderscore down operations.
- when \( a[\text{first}(p)\ldots\text{last}(p)] \cap (\text{key}(x)\ldots\text{down\textunderscore key}(x, l)) = \emptyset \) none of the subpackets of \( p \) will arrive to \((x, l)\).
- when \( a[\text{first}(p)\ldots\text{last}(p)] \cap (\text{key}(x)\ldots\text{down\textunderscore key}(x, l)) \neq \emptyset \) at most two subpackets of \( p \) go through \((x, l)\) at any step.

At any stage at most two packets arrive to \((x, l)\) and at most three packets are at that node/level.

Sketch of proof. The first two cases easily follow from the definition of zig_zag path. The third case has three subcases. When \( a[\text{first}(p)] \leq \text{key}(x) < a[\text{last}(p)] \leq \text{down\textunderscore key}(x, l) \) consider a subpacket \( q \) of \( p \) such that \( \text{first}(q) \leq \text{key}(x) < \text{last}(q) \) splitting in some node/level \((x', l')\) on the zig_zag path. Let \( q_1' \) and \( q_2' \) the two packets resulting from the split of \( q \), and assume that \( \text{key}(x) < \text{last}(q_1') \). Then the packet \( q_1' \) remains in \((x', l')\) but \( q_2' \) will move suffering no further splits to \((x, l)\). Therefore, it does not exist any subpacket of \( q_1' \) arriving to \((x, l)\) at the same moment that \( q_2' \). The argument is the same if \( \text{first}(q_2') \leq \text{key}(x) \). At most only one packet arrives to \((x, l)\) at any moment. The second subcase arises if \( \text{key}(x) < a[\text{first}(p)] \leq \text{down\textunderscore key}(x, l) < a[\text{last}(p)] \) and is analogous to the first subcase. The final subcase is \( a[\text{first}(p)] \leq \text{key}(x) < \text{down\textunderscore key}(x, l) < a[\text{last}(p)] \). Then, at some stage, a subpacket of \( p \) will split giving two packets satisfying the last two preceding conditions. As every subpacket of these can give at most one packet, we will get at most two subpackets arriving to \((x, l)\).

We now analyze the number of stages that the search procedure makes before all packets have been routed to their final destinations. Algorithm 1 checks the presence or absence of active packets after the execution of each stage. This introduces an undesirable \( \log k \) factor in the cost of the algorithm. Therefore, the search algorithm must be slightly changed, diminishing the number of times that the costly test is done. The desired performance is achieved executing \( \text{runs} \) of \( \lceil L(n) \rceil + \lceil \log_2 k \rceil \) stages, where \( L(n) = \log_{\frac{1}{\rho}} n \). After each run, we test whether there remains any active packet or not. If there is a least an active packet in \( P \), then a new run of \( \lceil L(n) \rceil + \lceil \log_2 k \rceil \) stages is executed, etc. As we shall soon prove in Lemma 3.3, the expected number of stages is \( O(\log n + \log k) \); hence, the expected number of runs is constant and the total expected cost is \( O(\log n + \log k) \).

For any two random variables \( X \) and \( Y \), we say that \( Y \) is a stochastic upper bound for \( X \) if and only if, for any \( t \), \( \Pr\{X > t\} \leq \Pr\{Y > t\} \). We shall write then \( X \leq_{\text{prob}} Y \). Note that \( X \leq_{\text{prob}} Y \) implies \( \mathbb{E}(X) \leq \mathbb{E}(Y) \). We will use \( B(n, p) \) to denote a random variable with binomial distribution and \( \text{NB}(r, p) \) to denote a random variable with negative binomial distribution. In both cases, \( p \) is the probability of success in a trial. Finally, in order to simplify the proof of our next Lemma, we assume w.l.o.g. that both \( L(n) \) and \( \log_2 k \) are integer numbers.

Lemma 3.3. The expected number of stages done by the search procedure before all packets become inactive is \( O(\log n + \log k) \), where \( n \) is the size of skip list \( n \) and \( k \) the number of keys to be searched.
Proof. Let $C_i$ denote the random variable whose value is the number of push\_forward, push\_down and split operations where key $a[i]$ gets involved before it reaches its final destination in the skip list of $n$ items. Let $C = \max_i \{C_i\}$. Clearly, $C$ is the number of stages that the parallel search makes before all packets become inactive.

We get an stochastic upper bound for $C$ deriving independent bounds for each of the contributions to the cost.

The number of push\_down operations applied to packets containing any given key is bounded by the level, level($S$), or height of the skip list. The expectation of the height for skip lists of size $n$ is $\Theta(L(n))$ (see for instance [4]).

Similarly, the number of push\_forward operations applied to packets containing any given key is bounded by the width of the skip list. The width $W(S)$ of a skip list $S$ is the maximum number of forward keys on any of its zigzag paths. It can also be defined as the maximum horizontal search cost in $S$ [3] or the maximum number of full horizontal steps on the path from the header to any item in $S$ [4]. Let $W$ denote the random variable corresponding to the width of skip lists of size $n$. An stochastic bound for $W$ is

$$W \leq \text{prob} \left( B\left(n, \frac{1}{np}\right) + \text{NB}(L(n) - 1, p) \right).$$

This bound is given in [6], although it is not presented in the same way as here.

Finally, the number of split operations where a key gets involved never exceeds $\log_2 k$, since each split halves the size of the packet containing that key.

Therefore, $E(C) \leq O(\log n) + E(W) + \log_2 k = O(\log n + \log k).$ 

4 Insertion and deletion

Assume, for the sake of simplicity, that the $k$ keys to be inserted are not already present in the skip list. The insertion algorithm has three main phases. First, it produces what we call an skip array by means of procedure make\_skip\_array (see Algorithm 2). Secondly, it makes a search of the $k$ keys using a slight variation of the procedure search of the previous section. Finally, the procedure merge inserts $k$ new nodes containing the $k$ keys in the appropriate places of the skip list.

To construct the skip array for the $k$ keys stored in array $a$, a random level is assigned to each key in the very first step. The level associated to each key will be the level of its corresponding node in the skip list. For all keys, in parallel, we make a call to function random\_level. The value it returns follows a $\text{NB}(1, p)$ distribution (see [6]). After an expected number $O(\log k)$ of iterations, the level of each key has been assigned. Let level[i] denote the level of key $a[i]$.

Then, the procedure make\_skip\_array allocates $k$ new nodes, keeping an array of pointers to these nodes (node), and putting each key and level at its correspondent node. The parallel dynamic memory manager needs $O(\log k)$ steps to allocate the $k$ new nodes [2].

The last step fills the skip array called Suc. The skip array Suc is a bidimensional array with $m = \max\{\text{level}[i] | 1 \leq i \leq k\}$ rows and $k$ columns. Each entry of array Suc, say Suc[i, l], is the minimum index $j$ to the right of $i$ ($i < j$) such that level[j] $\geq l$, if level[i] $\geq l$. If level[i] $< l$ or such an index does not exist then Suc[i, l] = 0. When Suc[i, l] $= j \neq 0$ we can say that node[j] will be the first new node of level $\geq l$ such that it is to the right of node[i]. In other words, in the list of level $l$, there will not be any new node between those containing $a[i]$ and $a[j]$. 


procedure make_skip_array (in a : array [1..k] of key_type; in/out Suc : skip_array.type; in p : real)
  var m : integer;
  begin
    m := max (level[i] | 1 ≤ i ≤ k);
    for all i : [1..k] do in parallel level[i] := random_level(p) end;
    for all i : [1..k] do in parallel Suc[i, l] := make_new_node(i, a[i], level[i]) end;
    for all l : l ∈ [1..m] do in parallel
      for all i : i ∈ [1..k] do in parallel
        if (level[i] ≥ l ∧ ∃ j : j > i ∧ i < j ∧ l ≤ level[j])
          Suc[i, l] := min {j | i < j ∧ l ≤ level[j]}
        else
          Suc[i, l] := 0
        end
      end
    end
  end make_skip_array

Algorithm 2: Procedure make_skip_array.

Computing the maximum m of the k levels can be done in $O(\log k)$ steps. On the other hand, each row of the skip array Suc can be filled independently. In order to fill one of these rows, say row l, we have just to perform a computation quite similar to prefix minimum.

Using $k/m$ processors for each row, it takes time $O(m)$ to fill all the rows of Suc. Since the expected value of m is $O(\log k)$, this part of the insertion process has expected cost $O(\log k)$ and uses $O(k)$ processors.

Once the skip array Suc has been built, the search procedure is called. Upon termination, each packet is between a pair of nodes of the skip list. All the keys in a given packet must be inserted at the list of level 1 between the node that the packet points to and its successor of level 1.

But in order to know the predecessors for each level of each of the new inserted nodes, we must also keep track of the path followed by the packets. Associated with each packet we will have an array update, such that update[l] is the rightmost node of level ≥ l in the path from the header to the current node/level pointed by the packet [6]. Therefore, the lowest key in the packet with level ≥ l will be stored in some node whose predecessor at level l is update[l]. In particular, we will allocate one update array for each key and consider that the update array of a packet p is that corresponding to the first key in the packet.

We will use update[i, l] to denote the l-th component of the update array corresponding to the i-th key, a[i]. To maintain the update arrays, the procedure push_down is modified in such a way that each time it is used we also perform the following assignment:

$$update[\text{first}(p), \text{level}(p)] := \text{node}(p).$$

The final phase of the insertion procedure is to merge the skip array with the skip list (see Algorithm 3). First, if $m = \max \text{level}(Suc)$ is greater than level($S$), we fill with pointers to the header node of $S$ the array update of each packet. After that, procedure brst_update broadcasts the update array of the first key of each packet to the other keys in the packet. Only the components 1 to m of the update arrays are needed and, since $m = O(\log k)$, the broadcasting can be done in expected time $O(\log k)$ using $O(k)$ processors.
Finally, we insert the new nodes between the nodes of the skip list, independently for every level by means of the procedure merge_level. The procedure merge can be executed with \(O(k)\) processors in expected time \(O(\log k)\) since the procedure merge_level has cost \(O(1)\) and we call it \(m\) times, once for each level to be updated.

```plaintext
procedure merge (in Suc: skip_array_type; in P: set_of_packets; in/out S: skip_list)
begin
  for all \(p : p \in P\) do in parallel
    for all \(l : l \in \text{level}(S)\) do in parallel
      update[first(p), l] := header(S)
    end
  end;
  brent_update(P);
  for \(l : l \in [1..\text{max_level}(Suc)]\) do merge_level(Suc, P, S, l) end;
end

procedure merge_level (in Suc: skip_array_type; in P: set_of_packets;
                        in/out S: skip_list; \(l : [1..k]\))
begin
  for all \(p : p \in P\) do in parallel
    for all \(i : (\text{first}(p..\text{last}(p))\) do in parallel
      if \(Suc[i, l] \neq 0 \land \text{update}[i, l] = \text{update}[Suc[i, l], l]\)
        forward(node[i, l] := node[Suc[i, l]]
      \[ Suc[i, l] \neq 0 \land \text{update}[i, l] \neq \text{update}[Suc[i, l], l]\)
        forward(node[i, l] := forward(update[i, l], l);
        forward(update[Suc[i, l], l, l] := node[Suc[i, l]]
      \[ Suc[i, l] = 0 \land \text{level}[i] \geq l \]
        forward(node[i, l] := forward(update[i, l, l]);
    fi
  end
end
end merge_level
```

Algorithm 3: Procedures merge and merge_level.

Consider a given level \(l\) and some new node, \(\text{node}[i]\), that stores \(a[i]\) and whose level is \(\text{level}[i] \geq l\). To insert \(\text{node}[i]\) in the linked list of level \(l\) we will need to know both its predecessor and its successor for that level. In the easiest case, these are given by \(\text{update}[i, l]\) and \(\text{forward}([\text{update}[i, l], l]\), respectively. But it can happen that the predecessor, the successor or both are nodes not yet inserted in the skip list. Let \(\text{Suc}[i, l] = j\). Assume that \(j \neq 0\). Then the successor at level \(l\) of \(\text{node}[i]\) is either \(\text{forward}(\text{update}[i, l], l)\) or \(\text{node}[j]\). If \(\text{update}[i, l] = \text{update}[j, l]\) then \(\text{node}[i]\) has to be the predecessor of \(\text{node}[j]\) and conversely, \(\text{node}[j]\) is the successor of \(\text{node}[i]\).

If \(j = 0\) then no new node could be the successor of \(\text{node}[i]\). Hence, the successor of \(\text{node}[i]\) at that level is \(\text{forward}(\text{update}[i, l], l)\).

We can perform the tests and change pointers in parallel, since only local information provided by the \text{update} and \text{Suc} arrays is needed. There is only one problem that we have
not yet considered: how to update the pointer of the nodes that are the predecessors of the first new nodes of each level. One possible way to solve that problem is to include a header in the skip array. This header could be considered as the 0-th node, having as many as \( m = \max_{\text{level}}(S_{\text{uc}}) \) levels, has not to be inserted in \( S \) and its update array points to NIL for all its levels. Nevertheless, this technique is not reflected in Algorithm 3.

Now, we briefly describe the deletion algorithm. It has two main phases. First, it searches the keys using the modified search procedure as in the insertion algorithm. We shall assume that the keys to be deleted are in fact present in the skip list. After this first phase, we have an update array for each key (using \texttt{brcst_update}) and a reference to each of the nodes that contain the given keys. The second phase of the algorithm actually removes the nodes. It begins constructing an skip array \( S_{\text{uc}} \) like in the insertion algorithm, but now the levels are not randomly assigned but the actual levels of the nodes.

For each node to be removed and each of its levels, we must elucidate whether: 1) it lies between two nodes that will remain; 2) its predecessor will be removed; 3) it successor will be removed; 4) both its predecessor and its successor will be removed. Which one of the four cases holds can be checked in parallel using the update and \( S_{\text{uc}} \) arrays. Cases 1 and 4 give no further problems. Cases 2 and 3 can be managed with simple parallel address arithmetic techniques similar to those used in the insertion algorithm. The performance of the insertion and deletion algorithms is given by the next Theorem.

**Theorem 4.1.** The insertion and deletion algorithms can be implemented in a EREW PRAM model with expected time \( O(\log n + \log k) \) and using \( O(k) \) processors, where \( k \) is the number of keys to be inserted or deleted and \( n \) is the length of the skip list.

**Acknowledgements**

We thank Torben Hagerup for suggesting us this research topic.

**References**


List of research reports (1993).

LSI-93-1-R “A methodology for semantically enriching interoperable databases”, Malú Castellanos.

LSI-93-2-R “Extraction of data dependencies”, Malú Castellanos and Félix Saltor.


LSI-93-4-R “An integral geometry based method for fast form-factor computation”, Mateu Sbert.

LSI-93-5-R “Temporal coherence in progressive radiosity”, D. Tost and X. Pueyo.

LSI-93-6-R “Multilevel use of coherence for complex radiosity environments”, Josep Vilaplana and Xavier Pueyo.

LSI-93-7-R “A characterization of PF^{NP[\Sigma]}_P = PF^{NP[\log]}_P”, Antoni Lozano.

LSI-93-8-R “Computing functions with parallel queries to NP”, Birgit Jenner and Jacobo Torán.


LSI-93-10-R “Parallel approximation schemes for problems on planar graphs”, Josep Díaz, María J. Serna, and Jacobo Torán.

LSI-93-11-R “Parallel update and search in skip lists”, Joaquim Gabarró, Conrado Martínez, and Xavier Messeguer.

Internal reports can be ordered from:

Nuria Sánchez
Departament de Llenguatges i Sistemes Informàtics (U.P.C.)
Pau Gargallo 5
08028 Barcelona, Spain
secrelsi@lsi.upc.es