PARALLEL UPDATE AND SEARCH IN SKIP LISTS*

Joaquim Gabarró, Conrado Martínez, and Xavier Messeguer

Departament de Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
Pau Gargallo 5, 08028-Barcelona, Spain
Contact e-mail: conrado@lsi.upc.es

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 simple 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 $O(\log n + \log k)$. The probability that there is a significant deviation from the expected time decreases as $O(n^{-2})$.

1 INTRODUCTION

A skip list is a randomized data structure that can be used for representing abstract data types such as dictionaries and ordered lists. Skip lists were introduced by W. Pugh in 1990[11] 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 as that of balanced trees. Moreover, the probability of the search time or space complexity exceeding their expected values, rapidly approaches 0 as the number of items in the skip list increases[12].

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 item and no balance, priority or weight information.

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

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[9], by L. Devroye[2], and by P. Kirschenhofer and H. Prodinger[8].

In a previous work, W. Paul, U. Vishkin and H. Wagener developed and analyzed the algorithms to dynamically maintain a parallel dictionary on 2–3 trees[10]. They considered the EREW PRAM model. Search, insertion and deletion algorithms for k items in a 2–3 tree storing n items were shown to take time $O(\log n + \log k)$ with $O(k)$ processors in the worst-case. The algorithms corresponding to the insertion and deletion are quite sophisticated.

Also, it is possible to design a parallel dictionary using hashing. This is a very active research field. In the case of randomized CRCW PRAM model, M. Dietzfelbinger and F. Meyer auf der Heide[4] implemented an optimal parallel dictionary with p processors in such way that n instructions (insert, delete, lookup) can be executed in optimal expected time $O(n/p)$. These results were improved using parallel dynamic hashing in real time[3]. Also, J. Gil, Y. Matias and U. Vishkin implemented a "nearly-constant" time optimal random NC parallel dictionary[6]. Finally H. Bast, M. Dietzfelbinger, T. Hagerup construct what they call a perfect dictionary, a scheme that allows p processors implement a set M in space proportional to $|M|$ to process batches of p insert, delete, and lookup instructions on M in constant time with high probability[1].

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. The profit of this dual view is the development of clear and easy algorithms.

In section 3 we present the algorithm to search for k keys in parallel. To derive it, we use the skip list as a tree. We assumme that we are given an ordered array of k distinct keys $a[1], a[2], \ldots, a[k]$. The algorithm routes a set of packets containing the keys along the skip list. Each packet holds a segment $a[i], a[i+1], \ldots, a[j]$ of the keys. We will say that a packet containing the keys $a[i] \leq a[i+i] \leq \cdots \leq a[j]$ *collides* with a key b when $a[i] \leq b < a[j]$. If a packet collides with a key, it is split into two packets; otherwise, the packet can be moved forward or down in the skip list. 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 arrays) 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 the work by Gabarró and Gavaldà[5]. 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 the performance of those for 2–3 trees[10].

Theorem 1.1. *Search, insertion and deletion algorithms to dynamically mantain a skip list can be implemented in the EREW PRAM model with expected time* $O(\log n + \log k)$ *and* k *processors, where* k *is the number of given keys or items and* n *is the number of items in the skip list. Moreover, the probability that the actual time of these algorithms significantly deviates from the expected time decreases as* $O(n^{-2})$.

We think these algorithms are important because they fulfil (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.

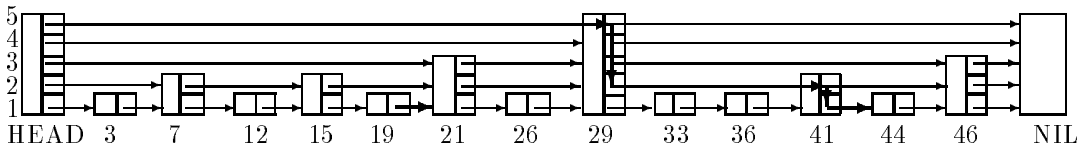

Figure 1: A skip list of 13 items and the search path to the interval (44,46].

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 identify the items with their keys. We give only an informal description of skip lists, more details can be found in Pugh's article[11]. A skip list contains several ordered linked lists. All the items are stored in one of the lists, the list of level 1. Some of the items belong also to the list of level 2; to decide which items should be included in the list of level 2 a random device is used. Once again, some of the items of the list of level 2 may belong to list of level 3, and so forth (see Figure 1).

Each item x in S has a key denoted as key(x). As yet mentioned, the items are sorted by key. A positive integer, denoted by level(x), is associated to each item x. If the level of a item x is level(x) = l, that means that x is included in the linked lists of level 1, 2, ..., l. The level of each item is given by a geometrically distributed random variable with parameter p. More precisely,

$$\Pr\{\text{level}(x) = k\} = p^{k-1}(1-p).$$

To implement a 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 its level indicates. These pointers, called *forward* pointers, point to the successors of the item in each list. If $0 \leq l \leq$ level(x), forward(x, l) denotes the succesor at level l of the node x. We also allocate a header node with pointers to each of the first nodes of each linked list. We shall write level(S) to denote the number of linked lists in the skip list S; level(S) is the maximum level of any of its items. The level of a skip list S is also called the *height* of the skip list. Also, we will use header(S) to denote the header node of S.

Moreover, we use the following conventions: 1) there is a node called **NIL** pointed to by the last node of each of the linked lists. By convention, level(**NIL**) = level(S)+1 = level(header(S))+1, and **NIL** is given a key greater than any legal key: key(**NIL**) = ∞; 2) the header node stores a dummy key that is less than any legal key: key(header(S)) = $-\infty$; 3) each item in a skip list S has a ficticious level 0; for all non-nil nodes x in S, forward(x, 0) = forward(x, 1); 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 item of the skip list S and l be some positive integer such that $l \leq$ level(x). The first node to the right of x with level $\geq$ 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 interval $S(x, l)$ as follows:

$$S(x, l) = (\text{key}(x), \text{key}(\text{wall}(x, l))].$$

If some legal key a belongs to $S(x, l)$, then it must be the key of a node or between the keys of two consecutive nodes, for some of the nodes between x and wall(x, l).

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

If $a \in S(x,0)$ then we can easily check if a is present or not in S: $a \in S \equiv (a = \text{key}(\text{forward}(x,1)))$. If $a \notin S$ but $a \in S(x,l)$, then x would be the predecessor (at level 1) of a node holding the key a. Note that the sequential search algorithm for skip lists just iterates while mantaining the invariant $a \in S(x,l)$, where a is the key to be searched for[11].

3 SEARCH

Given a 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 pointers called *node*, such that $node[i]$ points to the rightmost node x of skip list S having key such that $\text{key}(x) < a[i]$. Formally:

$$\begin{aligned} node[i] \;=\; & \text{a pointer to the unique node } x \text{ in } S \\ & \text{such that } \text{key}(x) < a[i] \leq \text{key}(\text{forward}(x,1)). \end{aligned}$$

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

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

The search algorithm routes a set of packets P along the data structure. This approach has been developed in the case of 2–3 trees by W. Paul, U. Vishkin and H. Wagener[10]. In more detail, the search algorithm proceeds in stages. At any given stage, the set of packets P must give us a full partition of the array $a[1..k]$; we must make sure that each key is "held" 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. In each step of the algorithm, each active packet is routed by moving it forward (changing the node it points to), moving it down (changing its level) or splitting it into two packets.

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 packets that are not active are those that have reached their final destination.

The subset of active packets in P is denoted $\text{active}(P)$. A processor must be associated with each active packet to perform the movement or split of each packet in parallel (see Algorithm 1).

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. At 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 mantain the following invariant: for each key a within p, $a \in S(\text{node}(p), \text{level}(p))$. The search algorithm finishes when $\text{active}(P)$ becomes empty.

```
procedure search(in a : array [1..k] of key_type; in S : skip_list; out node : array [1..k] of refnode_type)

    procedure route(in p : packet; in/out P : set_of_packets; in S : skip_list)
        var b : key_type;
            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)
            fi
        end route

    procedure split_and_push(in p : packet; in/out P : set_of_packets;
                             in S : skip_list)
        var b : key_type; m : 1..k; p1, p2 : packet;
            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)
              [] b ≥ a[last(p1)]  ⟶  push_down(p1, P, S)
            fi
        end split_and_push

    var p : packet; P : set_of_packets;
        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
        end;
        for all p : p ∈ P do in parallel
            for all i : first(p) ≤ i ≤ last(p) do in parallel
                node[i] := node(p)
            end
        end
end search
```

Algorithm 1: Search algorithm.

Let us consider an active packet p that points to (x, l) at stage t. Let b be the key of the succesor of node x at level l, that is, $b = \text{key}(\text{forward}(x, l))$ and let $b_W = \text{key}(\text{wall}(x, l))$.

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 interval $(b, b_W]$, provided that all them are in the interval $(\text{key}(x), b_W]$. More specifically, since $a \in S(x, l)$ for any key a in p by hypothesis, Lemma 2.1 implies the truth of $a \in S(\text{forward}(x, l), l)$. In that case, the packet must be pushed forward, replacing the node it points to by $\text{forward}(x, l)$.

2. If the last key in p is not greater than b ($b \geq a[\text{last}(p)]$), then for all keys a in p we have $a \in (\text{key}(x), b]$; that is, $a \in S(x, l-1)$ 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, i.e. b collides with the packet p. In that case, *route* calls procedure *split_and_push*. The packet $p =< [i..j], x, l, true >$ is halved into two packets $p_1 =< [i..m], x, l, true >$ and $p_2 =< [m+1..j], x, l, true >$, with $m = (i+j)$ **div** 2. The key b must not collide with at least one of p_1 and p_2. Therefore, one of the packets can be pushed forward or down applying one of the rules 1 and 2 above, while the other remains at the same node and level.

3.1 Efficiency of the Search Algorithm

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

Theorem 3.1. *In a skip list, the search algorithm 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 searched for and n is the length of the skip list. Moreover, the probability that the performance of the search algorithm deviates from the expected time decreases as $O(n^{-2})$.*

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 (x, l), in Lemma 3.2. This fact is needed to prove Lemma 3.1, although we shall not give here its proof.

In the second part, in Lemma 3.3, we give a bound on the expected number of iterations or stages that have to be performed in order to search for the k keys.

Theorem 3.1 directly follows from Lemmas 3.1 and 3.3.

Lemma 3.1. *The procedure route 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. We say that a packet p *crosses* a node/level (x, l) when:

- it arrives to (x, l) from a preceding node/level following a forward pointer or it arrives from $(x, l+1)$ going down one level, and
- it leaves (x, l) forwarded to the node/level $(forward(x, l), l)$ or steps down one level going to $(x, l-1)$.

Notice that all the packets crossing (x, l) must follow a unique labelled path that will be called $\text{zig_zag}(x, l) = [(\text{header}(S), \text{level}(S)) \xrightarrow{b_1} \cdots \xrightarrow{b_t} (x, l)]$. Each pair $(x', l') \longrightarrow (x'', l'')$ and its label b in the $\text{zig_zag}(x, l)$ path means that, at some step, we should go from (x', l') to (x'', l'') in order to reach (x, l) and that b is the key of x''. Consider any label b_i in the $\text{zig_zag}(x, l)$ path. If $b_i \leq \text{key}(x)$ the path "goes through" a forward pointer and b_i is said to be a *forward key*, otherwise if $b_i > \text{key}(x)$, the path "steps down one level" on the same node and b_i is said to be a *down key*. The keys in the zig_zag path to any node/level (x, l) can be partitioned into forward keys $f_1, f_2, \ldots, \text{key}(x)$ and down keys $d_1, d_2, \ldots, \text{down_key}(x, l)$ with $f_1 < \cdots < \text{key}(x) < \text{down_key}(x, l) \leq \cdots \leq d_1$. Therefore, any zig_zagpath can be represented as:

$$\text{zig_zag}(x, l) = [f_1, \ldots, \text{key}(x), \text{down_key}(x, l), \ldots, d_1].$$

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

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

We write zig_zag$(44, 1) = [29, 41, 44, 46, 46, \infty, \infty]$. The forward keys are $29, 41, 44$, the down keys are $\infty, \infty, 46, 46$ and down_key$(44, 1) = 46$.
We show that no packet generates a lot of packets crossing (x, l) together.

Lemma 3.2. *Let p be a packet in some node/level on the* zig_zag(x, l) *path.*

- *when* $a[\text{first}(p)..\text{last}(p)] \subseteq (\text{key}(x)..\text{down_key}(x, l)]$ *the packet p never splits and will move through (x, l) with no delay executing only push_forward and push_down operations.*
- *when* $a[\text{first}(p)..\text{last}(p)] \cap (\text{key}(x)..\text{down_key}(x, l)] = \emptyset$ *none of the subpackets of p will arrive at (x, l).*
- *when* $a[\text{first}(p)..\text{last}(p)] \cap (\text{key}(x)..\text{down_key}(x, l)] \neq \emptyset$ *at most two subpackets of p go through (x, l) at any step.*

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

Sketch of proof. The first two possibilities easily follow from the definition of zig_zag path. The third one has three cases. When $a[\text{first}(p)] \leq \text{key}(x) < a[\text{last}(p)] \leq \text{down_key}(x, l)$ 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 case arises if $\text{key}(x) < a[\text{first}(p)] \leq \text{down_key}(x, l) < a[\text{last}(p)]$ and is analogous to the first case. The final case is $a[\text{first}(p)] \leq \text{key}(x) < \text{down_key}(x, l) < a[\text{last}(p)]$. Then, at some stage, a subpacket of p will split giving two packets satifying 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 *runs* of $\lceil \text{L}(n) \rceil + \lceil \log_2 k \rceil$ stages, where $\text{L}(n) = \log_{1/p} n$. After each run, we test whether there remains any active packet or not. If there is at least an active packet in P, then a new run of $\lceil \text{L}(n) \rceil + \lceil \log_2 k \rceil$ 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 $\text{E}(X) \leq \text{E}(Y)$. We will use $\text{B}(n, p)$ to denote a random variable with *binomial distribution*. It is equal to the number of successes seen in a

series of n independent random trials, where the probability of success in a trial is p. We denote a random variable with *negative binomial distribution* as $\mathrm{NB}(r, p)$. It is equal to the number of failures seen before the r–th succes in a series of random independent trials, where the probability of success in a trial is p. Finally, in order to simplify the proof of our next Lemma, we assume w.l.o.g. that $\log_2 k$ is an 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 for.*

Proof. Let C_i denote the random variable whose value is the 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 $C_{n,k} = \max_{1 \le i \le k}\{C_i\}$. Clearly, $C_{n,k}$ is the number of stages that the parallel search makes before all packets become inactive, when searching for k keys in a skip list of size n. We get a stochastic upper bound for $C_{n,k}$ 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 level(S), i.e. by the height of the skip list. The expectation of the height[9] for skip lists of size n is $\Theta(\mathrm{L}(n))$.

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 $\mathrm{W}(S)$ of a skip list S is the maximum number of forward keys on any of its zig_zag paths. It can also be defined as the maximum number of forward pointers that should be followed to reach a node/level $(x, 0)$. The number of forward pointers to reach a given node/level $(x, 0)$ is called *horizontal search cost*[8] and *full horizontal steps*[9]). Let W_n denote the random variable corresponding to the width of a random skip list of size n. A stochastic bound for W_n is

$$W_n \le_{\mathrm{prob}} \mathrm{B}(n, \frac{1}{np}) + \mathrm{NB}(\mathrm{L}(n) - 1, p).$$

The bound is given in the work by Pugh[11], although it is not presented in the same way as here. The results obtained by Devroye[2] also confirm that $E(W_n) = \Theta(\log n)$.

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, $E(C_{n,k}) \le O(\log n) + E(W_n) + \log_2 k = O(\log n + \log k)$. ■

Using Chernoff bounds we can prove the following (like in the article by Sen[12]):

Lemma 3.4. *There exist constants γ_p and ϵ_p such that $\Pr\{C_{n,k} \ge \gamma_p \mathrm{L}(n) + \lceil \log_2 k \rceil\} \le \frac{\epsilon_p}{n^2}$. In other words, the probability that the search time deviates significantly from the expected time decreases as $O(n^{-2})$.*

For instance when $p = 0.5$ we can take $\gamma_p = 3.14$ and $\epsilon_p = 84.1$

4 INSERTION AND DELETION

Assume, for the sake of simplicity and without lost of generality, that we want to insert k distinct items not already present in the skip list. The insertion algorithm has four main phases (see Algorithm 2). First, the procedure *search_with_update* makes a search of the k keys using a slight variation of the procedure *search*. Second, *create_new_nodes* gives us k new nodes to hold the items to be inserted. Third, *make_succ* produces what we call an *skip array*, named *succ*. This array *succ*, together with the

information collected during the search, provides information about the predecessor and successor of each node/level to be inserted. Fourth, the procedure *merge* inserts the k new nodes containing the k items in the appropriate places of the skip list.

```
procedure insert (in a : array [1..k] of key_type; in/out S : skip_list)
    type skip_array  = array [0..k+1, 1..M] of 0..k;
         update_array = array [1..k, 1..M] of refnode_type;
         node_array  = array [0..k+1] of refnode_type;
         level_array  = array [0..k+1] of 1..M;

    procedure search_with_update(...);
    procedure create_new_nodes(...);

    procedure make_succ(in level : level_array; in m : 1..M; out succ : skip_array)
        var i : 0..k; l : 1..M;

            level[0] := m;
            level[k+1] := m;
            for all l : 1 ≤ l ≤ m do in parallel
                for all i : 0 ≤ i ≤ k ∧ level[i] ≥ l do in parallel
                    succ[i,l] := min(j : i < j ≤ k+1 ∧ level[j] ≥ l)
                end
            end
        end make_succ

    procedure merge(...);

    var succ : skip_array;
        update : update_array;
        node : node_array;
        level : level_array;
        m : 1..M;

        search_with_update (a, S, node, update);
        create_new_nodes(a, node, level);
        m := max(level[i] : 1 ≤ i ≤ k);
        make_succ(level, m, succ);
        merge(succ, update, node, m, S)
    end insert
```

Algorithm 2: Procedure insertion.

Procedure search_with_update: Upon termination each packet p has a predecessor in the skip list S at each level l. To know it, we must keep track of the path followed by p. Hence, for each packet p, we take an array $update$[11], such that:

$$update_p[l] \;=\; \text{rightmost node of level} \geq l \text{ in the zig_zag path from the header to the current node/level pointed to by } p.$$

Later, we spread the update arrays of the packets to all the keys. If a key $a[i]$ was in packet p when p had level l, then the l-th component of the update array should be copied to the update array associated with $a[i]$. From now on, we consider that the update array associated to a packet is the update array associated with the first key in the packet. We shall write $update[i,l]$ to denote the l-th component of the update array corresponding to the key $a[i]$. To maintain the $update$ arrays, the procedure *push_down*

is modified including the assignment:

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

Each level of the update arrays can be optimally spread using $O(\log k)$ time and $O(k/\log k)$ processors. By the slowdown principle, we can allocate $k/\text{level}(S)$ processors to each level of S, and hence the broadcasting of the update arrays can be done in time $O(\text{level}(S))$ using k processors. Since the expected level of S is $O(\log n)$ the expected time to spread the update arrays is $O(\log n)$ using $O(k)$ processors.

Procedure create_new_nodes: When creating the new nodes 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 NB$(1, p)$ distribution[11]. 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 $a[i]$.

Finally, the procedure allocates k new nodes, keeping an array of pointers to these nodes ($node$), and puts each key and level at its corresponding node. The parallel dynamic memory manager needs $O(\log k)$ steps to allocate the k new nodes[7].

Procedure make_succ: It builds the skip array $succ$. This array mimics a new skip list correspoding to the keys in a. It is build on a two dimensional array with $m = \max\{level[i] : 1 \leq i \leq k\}$ useful rows and $k + 2$ columns. Any column i with $1 \leq i \leq k$ has $level[i]$ useful rows. Columns 0 and $k+1$ are sentinels with m rows. The array $succ$, for $0 \leq i \leq k$, is defined as:

$$succ[i, l] = \text{Minimum } j \text{ to the right of } i, \text{ that is, } i < j, \text{ such that } \ level[j] \geq l.$$

In other words, in the list of level l, there will not be any new node between those containing $a[i]$ and $a[j]$. Note that $succ[0, l]$ is the index of the first node to be inserted whose level is at least l. Computing m can be done in $O(\log k)$ steps. Moreover all the rows of $succ$ can be computed independently. In order to fill each of the rows we need only easy tree computations. Using k/m processors for each row, it takes time $O(m)$ to fill all the rows of $succ$. Since the expected value of m is $O(\log k)$, this part of the insertion has expected cost $O(\log k)$ with $O(k)$ processors.

Procedure merge: This phase merges $succ$ with S (see Algorithm 3). If $m >$ level(S), we fill with pointers to header(S) the rest of the arrays $update$. Later, we insert the new nodes level by level in S using the procedure *merge_level*. Let us consider it in more detail. Consider a level l and the node $node[i]$ corresponding to $a[i]$ with $level[i] \geq l$. To insert $node[i]$ in the linked list of level l we need to know both its predecessor and its successor for that level. There are four different cases (see Algorithm 3). We consider only one of them, others are similar. Given i with $1 \leq i \leq k$, call $succ[i, l] = j$ and suppose that

$$j \neq k + 1 \wedge update[i, l] \neq update[j, l].$$

As j is different from the sentinel $k + 1$ there exists a key $a[j]$ of level $\geq l$ to the right of $a[i]$. Moreover, as $update[i, l] \neq update[j, l]$ there exist nodes of S in level l whose keys lie between $a[i]$ and $a[j]$. Since $a[j]$ is not the successor of $a[i]$ at level l in the new skip list, the following assignments are needed:

$$\text{forward}(node[i], l) := \text{forward}(update[i, l], l);$$
$$\text{forward}(update[j, l], l) := node[j];$$

We can perform the tests and change pointers in parallel, since only the local information provided by the $update$ and $succ$ arrays is needed. The procedure *merge*

```
procedure merge (in succ : skip_array; in update : update_array;
                 in node : node_array; in m : 1..M; in/out S : skip_list)

    procedure merge_level (in succ : skip_array; in update : update_array;
                           in node : node_array; in/out S : skip_list; in l : 1..M)
        var i : 1..k; j : 0..k;
            for all i : 1 ≤ i ≤ k ∧ level[i] ≥ l do in parallel
                j := succ[i, l];
                if
                    j ≠ k + 1 ∧ update[i, l] = update[j, l] ⟶ forward(node[i], l) := node[j]
                  ▯ j ≠ k + 1 ∧ update[i, l] ≠ update[j, l] ⟶ forward(node[i], l) := forward(update[i, l], l);
                                                               forward(update[j, l], l) := node[j]
                  ▯ j = k + 1 ⟶ forward(node[i], l) := forward(update[i, l], l)
                fi
            end
            forward(update[succ[0, l], l], l) := node[succ[0, l]]
        end merge_level

    var l : 1..M;
        i : 1..k;
        for all l : level(S) ≤ l ≤ m do in parallel
            for all i : 1 ≤ i ≤ k do in parallel
                update[i, l] := header(S)
            end
        end;
        for l : 1 ≤ l ≤ m do
            merge_level(succ, update, node, S, l)
        end;
        level(S) := max(level(S), m)
    end merge
```

Algorithm 3: Procedures merge and merge_level.

can be executed with $O(k)$ processors in expected time $O(\log k)$ since *merge_level* has cost $O(1)$ using $O(k)$ processors and it is called m times, once for each level.

Now, we sketch the deletion algorithm. First, it uses the search with update procedure to find where are the keys to be deleted. After this, it constructs three skip arrays similar to the skip array *succ* used in the insertion procedure, but now their number of rows is computed from the actual levels of the nodes to be deleted. Now, for each node to be removed and each of its levels, we must know 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 the three skip arrays giving informations about sucessors, predecessors and groups of consecutive nodes to be deleted. 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. *In a skip list, 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. Moreover, the probability that the performance of the insertion and deletion algorithms significantly deviates from the expected time decreases as $O(n^{-2} + k^{-2})$.*

ACKNOWLEDGEMENTS

We thank Torben Hagerup to help us with this research.

REFERENCES

1. H. Bast, M. Dietzfelbinger, and T. Hagerup. A perfect parallel dictionary, in: "Proc. 17th Mathematical Foundations of Computer Science, LNCS 598," I. M. Havel and V. Koubek, eds., Springer-Verlag (1992).

2. L. Devroye. A limit theory for random skip lists, *The Annals of Applied Probability*, 2:597–609 (1992).

3. M. Dietzfelbinger and F. Meyer auf der Heide. Dynamic hashing in real time, in: "Informatik. Festschrift zum 60. Geburtstag von Günter Hotz. Teubner-Texte zur Informatik 1," J. Buchmann, H. Ganzinger, and J. W. Paul, eds., B. G. Teubner (1992).

4. M. Dietzfelbinger and F. Meyer auf der Heide. An optimal parallel dictionary, *Information and Control*, 102:196–217 (1993).

5. J. Gabarró and R. Gavaldà. An approach to correctness of data parallel algorithms, *J. Parallel and Distributed Computing* (1993).

6. J. Gil, Y. Matias, and U. Vishkin. Towards a theory of nearly constant time parallel algorithms, in: "Proc. 32nd Foundations of Computer Science," (1991).

7. W. D. Hillis and G. L. Steele, Jr. Data parallel algorithms, *Comm. ACM*, 29:1170–1183 (1986).

8. P. Kirschenhofer and H. Prodinger. The path length of random skip lists, Preprint. Technical University, Wien, (1993).

9. T. Papadakis, J. I. Munro, and P. V. Poblete. Analysis of the expected search cost in skip lists, in: "Proc. 2nd Scandinavian Workshop on Algorithm Theory, LNCS 447," J. R. Gilbert and R. Karlsson, eds., Springer-Verlag (1990).

10. W. Paul, U. Vishkin, and H. Wagener. Parallel computation on 2–3 trees, in: "Proc. 10th International Colloquium on Automata, Programming and Languages, LNCS 154," J. Díaz, ed., Springer-Verlag (1983).

11. W. Pugh. Skip lists: a probabilistic alternative to balanced trees, *Comm. ACM*, 33:668–676 (1990).

12. S. Sen. Some observations on skip-lists, *Inform. Proc. Lett.*, 39:173–176 (1991).