A Top-Down Design of a Parallel Dictionary using Skip Lists*

Joaquim Gabarró Conrado Martínez Xavier Messeguer†

12th April 1994

Abstract

We present a top down design of a parallel PRAM dictionary using 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 implement on real machines. 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 $O(n^{-2})$ in the search and as $O(n^{-2} + k^{-2})$ in the insertion and deletion.

1 Introduction

Parallel dictionaries are important data structures widely studied. In a systolic framework, priority queues and search trees were designed by C.E. Leiserson in [13]. Later, M.J. Atallah and S.R. Kosaraju [1] developped a generalized dictionary where a sequence of operations can be pipelined at constant rate.

In the PRAM context, W. Paul, U. Vishkin and H. Wagener developed and analyzed algorithms to dynamically maintain a parallel dictionary on 2–3 trees [16]. They considered the EREW PRAM model. Parallel 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 [6] implemented an optimal parallel dictionary with p processors in such way that n instructions (insert, delete, lookup) can be executed in optimal expected

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

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

time $O(n/p)$. These results were improved using parallel dynamic hashing in real time [5]. Also, J. Gil, Y. Matias and U. Vishkin implemented a "nearly-constant" time optimal random NC parallel dictionary [10]. 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 [2].

Parallel dictionaries have also been implemented on massively parallel machines. In a MIMD framework T. Duboux, A. Ferreira and M. Gastaldo in [7] have implemented in a Volvox IS860 machine from Archiple Co. a ring of dictionary machines. These machines use sequential algorithms on 2–3 trees as local data structures. Recently M. Gastaldo [9] has implemented a parallel dictionary in a SIMD machine. He has taken the Mas Par-1, also known as DECCmpp 120000. A linear array is used to represent local dictionaries in each processor. Assuming a load balancing of the local data structures in the processors, one key can be searched in time $O(\log n/k)$ and inserted or deleted in time $O(n/k)$ with k processors.

In our work we present a design of a parallel dictionary using skip lists. 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 [18] 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. As skip lists behave in some aspects like balanced trees and in some other aspects as linked lists, we profit of this dual view in the development of clear and easy algorithms.

In section 2 we review some basic facts about skip lists. We give the sequential search and insertion algorithms [18] in a notation close to the parallel version. Two fundamental ideas are introduced here, the definition of zig_zag path and that of packet.

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. Assumming that we are given an ordered array of k keys, the algorithm routes a set of packets containing the keys along the skip list. Special care is taken to obtain readable algorithms. We apply stepwise development techniques along the lines suggested in the work by Gabarró and Gavaldà [8] and L. Mounier and G. Utard [14]. This is fundamental to be able to implement these algorithms on real machines. From a theoretical point of view, we also obtain interesting results because the expected performance of our algorithms is comparable to the performance of those for 2–3 trees [16].

> A search algorithm to dynamically maintain a skip list can be obtained 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 $O(n^{-2})$.

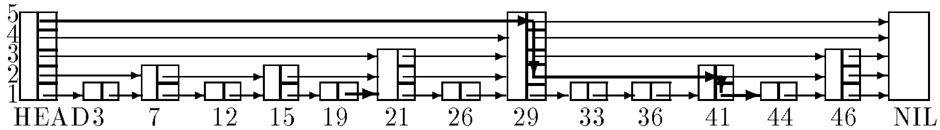

Figure 1: A skip list of 13 items and the search path to the interval (44,46].

In section 4 and section 5 we obtain algorithms for insertion and 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. Both algorithms are iterative. As before, the expected time is $O(\log n + \log k)$ and the deviation from this expected time decreases as $O(n^{-2} + k^{-2})$.

Finally, in Section 6 we discuss several implementation-related issues and give some hints for the implementation of our algorithms using C*®, the parallel data language for the Connection Machine.

Section 7 is devoted to conclusions.

2 Skip Lists

Skip lists are randomized data structures introduced by W. Pugh in 1990 [18]. They can be used to represent abstract data types such as dictionaries and ordered lists. 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. 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 [19]. They have a rich and interesting probabilistic analysis; consider, for instance, the work done by T. Papadakis, J. I. Munro, and P.V. Poblete [15], by L. Devroye [4], and by P. Kirschenhofer and H. Prodinger [12].

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 [18]. A non-empty skip list consists of several non-empty ordered linked lists. All the items are stored in one of the lists, namely 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 a skip list S (from now on S will denote a skip list) has a key denoted as key(x). As yet mentioned, the items are sorted by key in each list. 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 $1, 2, \ldots, l$. 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 $1 \le l \le \text{level}(x)$, forward($x, l$) denotes the succesor at level l of the node x. We also allocate a header node, called header(S), with pointers to the first node 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 among the levels of its items. The level of a skip list S is also called the *height* of the skip list. We will also use the following conventions:

i) 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**) $= \infty$;

ii) the header node stores a dummy key that is less than any legal key: key(header(S)) $= -\infty$;

iii) each item in a skip list S has a ficticious level 0; for all non-nil nodes x in S, forward($x, 0$) $=$ forward($x, 1$);

iv) each node stores as many copies of its key as its level indicates. This convention is redundant in the sequential algorithms, but it is needed to avoid read conflicts in the parallel case.

In order to carefully analyze the algorithms running on skip lists we define now a path between the initial node/level (header(S), level(S)) and a particular node/level (x, l). This path is a sequence of node/levels, and there is an unambigous way to label each pair of consecutive node/levels. We call it zig_zag(x, l).

Definition 2.1. *The path* $\text{zig\_zag}(x,l) = [(\text{header}(S), \text{level}(S)) \xrightarrow{b_1} \cdots \xrightarrow{b_t} (x,l)]$ *is defined recursively as follows:*

- *The recursive basis is* $\text{zig\_zag}(\text{header}(S), \text{level}(S)) = [\ (\text{header}(S), \text{level}(S))\ ]$.
- *If* $l < \text{level}(x)$, $\text{zig\_zag}(x,l) = [\text{zig\_zag}(x, l+1) \xrightarrow{b} (x,l))]$, *where the label is* $b = \text{key}(\text{forward}(x, l+1))$.
- *If* $l = \text{level}(x)$, $\text{zig\_zag}(x,l) = [\text{zig\_zag}(\text{backward}(x,l), l) \xrightarrow{b} (x,l))]$, *where* $b = \text{key}(x)$ *and* backward(x, l) *is the node* y *such that* forward(y, l) $= x$

We can omit the node/levels in the path and write $\text{zig\_zag}(x,l) = [b_1, \ldots, b_t]$.

Let us classify the keys in $\text{zig\_zag}(x,l) = [(\text{header}(S), \text{level}(S)) \xrightarrow{b_1} \cdots \xrightarrow{b_t} (x,l)]$ into two classes. 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'') and $b = \text{key}(\text{forward}(x',l'))$. When $b \leq \text{key}(x)$ the path "goes through" a forward pointer, (x',l') is called a *forward node/level* and b is said to be a *forward key*. Otherwise, $b > \text{key}(x)$ and the path "steps down one level" on the same node, (x',l') is called a *down node/level* and b is said to be a *down key*. The keys are then partitioned into forward keys $f_1, f_2, \ldots, f_l$ and down keys $d_1, d_2, \ldots, d_r$. It is easy to prove that $f_l = \text{key}(x)$. We define the down_key corresponding to (x,l) as d_r and we write $d_r = \text{down\_key}(x,l)$. Therefore $f_1 < \cdots < \text{key}(x) < \text{down\_key}(x,l) \leq \cdots \leq d_1$ and the zig_zag path 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 $\text{zig\_zag}(44,1) = [29, 41, 44, 46, 46, \infty, \infty]$. The forward keys are in this example $29, 41, 44$, the down keys are $\infty, \infty, 46, 46$. By definition $\text{down\_key}(44,1) = 46$.
Let x be a item of the skip list S and l be some integer such that $0 \leq l \leq \text{level}(x)$, we write

$$\text{wall}(x,l) = \text{"the first node } y \text{ to the right of } x\text{, i.e. } \text{key}(y) > \text{key}(x) \text{ such that level}(y) > l\text{"}.$$

In the preceding example $\text{wall}(\text{header}(S), 2) = 21$. Note that $\text{wall}(x,0) = \text{forward}(x,0)$ and $\text{key}(\text{wall}(x,l)) = \text{down\_key}(x,l)$. Let $S(x,l)$ be the following interval:

$$S(x,l) = (\text{key}(x), \text{down\_key}(x,l)] = \{k \mid \text{key}(x) < k \leq \text{down\_key}(x,l)\}$$

Therefore, if some key k 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 $\text{wall}(x,l)$.

Lemma 2.1. *Let S be a skip list and let a be a key such that for some item x and a level $l \geq 0$, $a \in S(x,l)$. Let b be the key stored at the succesor of x at level l, that is, $b = \text{key}(\text{forward}(x,l))$.*

- *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 because $a \in S \equiv (a = \text{key}(\text{forward}(x,1)))$.*
- *If $a \notin S$ but $a \in S(x,0)$, then x would be the predecessor (at level 1) of a node holding the key a.*

In the following we survey the sequential search and update algorithms. Our presentation differs slightly from the original one given by Pugh in [18] in order to emphasize the commonalities with their parallel counterparts.

```
procedure sequential_search(in a : key_type; in S : skip_list;
                            out node : refnode_type)

   procedure route(in p : packet; in S : skip_list)
      var b : key_type;

      {key(p) ∈ S(node(p), level(p)) ∧ active(p) ∧ active(p) ≡ (level(p) > 0)}

      b := key(forward(node(p), level(p)));
      if
         b < key(p)  ⟶  {key(p) ∈ S(node(p), level(p)) ∧ b < key(p)}⇒
                         {key(p) ∈ S(forward(node(p)), level(p))}

                         node(p) := forward(node(p), level(p))

       [] b ≥ key(p)  ⟶  {key(p) ∈ S(node(p), level(p)) ∧ b ≥ key(p)}⇒
                         {key(p) ∈ S(node(p), level(p) − 1)}

                         level(p) := level(p) − 1

      fi;
      active(p) := level(p) > 0

      {key(p) ∈ S(node(p), level(p)) ∧ active(p) ≡ (level(p) > 0)}
   end route

   var p : packet;

   {a ∈ S(header(S), level(S)) = (−∞, +∞]}

   p :=< a, header(S), level(S), (level(S) > 0) >;

   {key(p) ∈ S(node(p), level(p)) ∧ active(p) ≡ (level(p) > 0)}

   do  level(p) > 0  ⟶
       {key(p) ∈ S(node(p), level(p)) ∧ active(p) ∧ active(p) ≡ (level(p) > 0)}

       route(p, S)

       {key(p) ∈ S(node(p), level(p)) ∧ active(p) ≡ (level(p) > 0)}
   od;

   {key(p) ∈ S(node(p), level(p)) ∧ ¬ active(p) ∧ active(p) ≡ (level(p) > 0)}⇒
   {key(p) ∈ S(node(p), 0)}⇒
   {node(p) points to the node x such that key(x) < key(p) = a ≤ key(forward(x, 1))}

   node := node(p)

end search
```

Algorithm 1: Sequential search algorithm.

• *Sequential search algorithm:* Given a skip list S and a key a, the sequential search problem can be specified as follows:

$\{a \in \mathrm{S}(\mathrm{header}(S), \mathrm{level}(S))\}$
$\mathrm{search}(a, S, node)$
$\{node$ points to the node x in S such that
$\mathrm{key}(x) < a \leq \mathrm{key}(\mathrm{forward}(x, 1))\}$

In Algorithm 1 we recall the sequential search algorithm solving this problem. Intuitively, the search procedure works moving forward or down a through the skip list S until it reaches the unique node x such that $\mathrm{key}(x) < a \leq \mathrm{key}(\mathrm{forward}(x, l))$. In any given stage of the search, the key is said to be at a node/level (x, l) called the *current node/level*. Initially, the current node/level is $(\mathrm{header}(S), \mathrm{level}(S))$. The search procedure iterates mantaining the invariant $a \in S(x, l)$ until the current level is 0. It is useful to collect all the information about the key a, the current node/level (x, l) and finishing condition $(l > 0)$ in a "packet" p, such that

$$p =< a, x, l, (l > 0) >=< \mathrm{key}(p), \mathrm{node}(p), \mathrm{level}(p), \mathrm{active}(p) >$$

If p contains a reference to a node/level (x, l) at some stage t of the algorithm, we shall say that p *points to* (x, l) during the stage t. At each step of the algorithm p is routed through S moving it forward (changing the node it points to), or moving it down (changing its level), until the current level of p is 0 and p becomes incative. Then the current node in p points to the adecuate position.

Lemma 2.2. *The search procedure routes* $p =< a, \mathrm{header}(S), \mathrm{level}(S), (\mathrm{level}(S) > 0) >$ *through the skip list* S*. If at a given stage,* p *has* (x, l) *as a current node/level, the path followed by* p *is* $\mathrm{zig\_zag}(x, l)$.

In Algorithm 1 we give an annotated version of the sequential search. The procedure *route* gives us a move of p in S and verifies

$$\{\mathrm{key}(p) \in S(\mathrm{node}(p), \mathrm{level}(p)) \ \wedge \ \mathrm{active}(p)\} \ \mathrm{route} \ \{\mathrm{key}(p) \in S(\mathrm{node}(p), \mathrm{level}(p))\}$$

Note that, in order to prove the correctness we use the equivalence

$$\mathrm{key}(p) \in S(\mathrm{node}(p), 0) \equiv \text{ "node}(p) = \text{a pointer to the unique node } x \text{ in } S \text{ such that } \mathrm{key}(x) < \mathrm{key}(p) \leq \mathrm{key}(\mathrm{forward}(x, l))\text{"}$$

Finally, we state the following theorem due to Pugh [18] and Papadakis, Munro, Poblete [15] on the performance of the sequential search:

> Let $C_n \equiv C_{n,1}$ be the random variable denoting the cost of searching for an arbitrary key a in a random skip list of n items. The expected value of C_n is $\Theta(\log n)$.

```
{we assume that a is not in S }
procedure insert (in a : key_type; in/out S : skip_list)

  type update_array = array [1..M] of refnode_type;

  procedure search_with_update(...);
  procedure create_new_nodes(...);
  procedure merge(in update : update_array; in level : 1..M;
                  in node : refnode_type; in/out S : skip_list)
    var l : 1..M;


    {a = key(node) ∧ level = level(node) ∧
    ∀l : 0 < l ≤ level(S) : update[l] = u_l ∧
    key(update[l]) < a ≤ key(forward(update[l], l))}

    for l : level(S) < l ≤ level do
      update[l] := header(S)
    end;

    level(S) := max(level(S), level);
    ...

    for l : 1 ≤ l ≤ level do
      forward(node, l) := forward(update[l], l);
      forward(update[l], l) := node
    end

    {a = key(node) ∧ ∀l : 0 < l ≤ level : forward(update[l], l) = node ∧ forward(node, l) = u_l}

  end merge

  var update : update_array; node : refnode_type; level : 1..M

  search_with_update (a, S, update);

  {∀l : 0 < l ≤ level (S) : update[l] = u_l ∧ key(update[l]) < a ≤ key(forward(update[l], l))}

  create_new_node(a, node, level);

  {level(node) = level ∧ key(node) = a ∧ Pr{level = k} = p^(k-1)(1 - p)
  for some 0 < p < 1; moreover, ∀l : 0 < l ≤ level : update[l] = u_l ∧
  key(update[l]) < key (node) ≤ key(forward(update[l], l))}

  merge(update, level, node, S);

  {the key a has been inserted in S }

end insert
```

Algorithm 2: Sequential insertion algorithm.

• *Sequential insertion algorithm:* It is given in Algorithm 2. We assume, without lost of generality, that the key a to be inserted does not belong to S. It has three main phases coded in three different procedures. First, it is necessary to modify the search procedure to keep information about the nodes preceding the new one to be inserted. This was done by Pugh defining an array update. It is useful to enclose this array into p:

$$p = < \text{key}(p), \text{node}(p), \text{level}(p), \text{active}(p), \text{update}(p) >$$

where

$$\text{update}(p)[l] = \quad \text{"rightmost node of level} \geq l \text{ in the zig\_zag path from the header to the current node/level pointed to by } p\text{"}$$

The version of the search procedure computing the update array will be called *search_with_update*. Second, there is a procedure *create_new_node* to allocate a new node, called *node*, to be inserted in S. The procedure *create_new_node* calls a function *random_level()* giving a geometrically distributed random variable. This function is implemented in [18] mimicking the definition of a geometrical random variable, that is:

$$\text{level} := 1;\ \textbf{do}\ \text{random}() < p \rightarrow \text{level} := \text{level} + 1\ \textbf{od};\ \text{returns level}$$

Finally, the procedure *merge* updates the necessary links to insert *node* in the appropiate place.

• *Sequential deletion algorithm:* Goes along the same lines a insertion algorithm. It uses a procedure *remove* that changes the necessary links to remove the node from each of the linked lists of S.

```
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
  od;
  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 3: Search algorithm.

3 Search

The algorithm routes packets along the data structure. A similar approach has been developed in the case of 2–3 trees by W. Paul, U. Vishkin and H. Wagener [16]. To be more precise, given a skip list S of n items and an ordered array $a[1\ ..\ k]$ with k keys, $a[1] \leq a[2] \leq \cdots \leq a[k]$, the search algorithm returns an array of k pointers $node[1\ ..\ k]$, such that:

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

As in the sequential case, fundamental to our search algorithm is the notion of *packet*. Now a packet p stores a current node/level, and active status, and two indexes i and j, $1 \leq i \leq j \leq k$, representing the subarray $a[i\ ..\ j]$ of $a[1\ ..\ k]$. The subarray $a[i\ ..\ j]$ contains the keys to be searched for. We denote $\mathrm{first}(p) = i$ and $\mathrm{last}(p) = j$. More formally,

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

We denote by P the set of all the packets. This set must give us a full partition of the array $a[1..k]$ because we must make sure that each key is "held" by one and only one packet. A given packet p can be active or inactive. It is active if its level is not null, i.e. $\mathrm{active}(p) \equiv (\mathrm{level}(p) > 0)$. The subset of active packets in P is denoted $\mathrm{active}(P)$. The packets that are not active are inactive. At the very start of the algorithm a unique active packet containing all k keys is "injected" into the beginning of the skip list S by making $(\mathrm{header}(S), \mathrm{level}(S))$ the current node/level. Later, the search algorithm proceeds in stages routing the set of active packets along the skip list S. In each stage, each active packet is routed by moving it forward, down, splitting it into two packets. At the end of a step a packet can become inactive. Finally all the packets become inactive. The packets that are inactive are those that have reached their final destination. A processor must be associated with each active packet to perform the movement or split of each packet in parallel (see Algorithm 3).

• Procedure *route*: At each stage, all active packets are routed through the skip list S. The procedure *route*(p,P,S) moves or splits p in order to mantain the invariant:

$$a[\mathrm{first}(p)\ .\ .\mathrm{last}(p)] \subseteq S(\mathrm{node}(p), \mathrm{level}(p)).$$

The main loop containing *route* finishes when $\mathrm{active}(P)$ becomes empty. Once all packets become inactive, the key of the current node has to be spread to its keys. The procedure *route* pushes forward, down or splits the packet p using the procedures *push_forward*, *push_down* and *split_and_push*.

• Procedure *push_forward*: If the first key in p is greater than $\mathrm{key}(\mathrm{forward}(x, l))$, then all the keys in p must be in the interval $S(\mathrm{forward}(x, l), l)$, provided that all them are in the interval $S(x, l)$ by Lemma 2.1. By hypotesis, this is indeed the case, and the packet must be pushed forward, replacing x by forward(x, l). We next give the formal specification of the procedure *push_forward*. We should have the skip list S as a parameter in the specification in order to it being properly defined but we will omit S to simplify the notation.

$$\{P = Q \ \wedge \ p \in P \ \wedge \ p =< [i..j], x, l, true > \ \wedge 1 \leq l \leq \text{level}(x)\}$$

$$\text{push\_forward}(p, P, S)$$

$$\{P = p' \cup Q \setminus p \ \wedge \ p' =< [i..j], \text{forward}(x, l), l, true >\}$$

• Procedure *push_down*: Assume the last key in p is not greater than key(forward(x, l)), i.e. key(forward$(x, l)) \geq a[\text{last}(p)]$. Then for all keys a in p it holds that $a \in S(x, l-1)$, and the packet must be pushed down, decrementing its level. If the level of p becomes 0, then the packet is made inactive, so it will not be routed in the next stages. More formally,

$$\{P = Q \ \wedge \ p \in P \ \wedge \ p =< [i..j], x, l, true > \ \wedge 1 \leq l \leq \text{level}(x)\}$$

$$\text{push\_down}(p, P, S)$$

$$\{P = p' \cup Q \setminus p \ \wedge \ p' =< [i..j], x, l-1, (l \geq 1) >\}$$

• Procedure *split_and_push*: Finally, $b = \text{key}(\text{forward}(x, l))$ could fall in the midst of the first and last key of the packet, i.e. $a[\text{first}(p)] \leq b < a[\text{last}(p)]$. 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, using *split_packet*. This procedure can be specified as:

$$\{P = Q \ \wedge \ p \in P \ \wedge \ p =< [i..j], x, l, true > \ \wedge i \leq m < j\}$$

$$\text{split\_packet}(p, m, p_1, p_2, P, S)$$

$$\{P = \{p_1\} \cup \{p_2\} \cup Q \setminus p \ \wedge \ p_1 =< [i..m], x, l, true > \ \wedge p_2 =< [m+1..j], x, l, true >\}$$

The key $b = \text{key}(\text{forward}(x, l))$ 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 for the procedures above, while the other remains at the same node and level.

3.1 Contention analysis

Recall that during a routing step all the active packets execute the statement $b := \text{key}(\text{forward}(x, l))$. Hence, read conflicts become possible if more than one packet is located in a nodel/level (x, l). We have not adressed this problem before. It is clear that a routing step takes time $O(1)$ if run in a CREW model. It will be interesting to find a EREW implementation of the route procedure with the same amount of resources. To do it we need:

- to bound the number of packets in a node/level by a constant. Therefore read conflicts are also bounded.
- to give a protocol allowing different packets read without conflict. We distinguish the possibly conflicting packets taking into account the different ways that a packet crosses a node/level.

We define the predicate *remains* such that:

t=0: $\text{remains}_0(p)$=false

t>0: $\text{remains}_t(p)$ is true if p has been generated by an split on node/level (x, l) at stage t and has remained on this same node/level (x, l) at the end of the stage; otherwise is false

The second one is a function $\text{split\_side}(p) \in \{\text{left}, \text{right}\}$. At the beginning of the search algorithm the packet having all the keys is located in $(\text{header}(S), \text{level}(S))$ of the skip list S. As this packet is not the result of a split operation we can define its split_side either left or right.

t=0: $\text{split\_side}_0(p) = \text{left}$

t>0: If p has not just been split $\text{split\_side}_t(p) = \text{split\_side}_{t-1}(p)$. Otherwise, p has been generated by an split from a packet q. If $\text{first}(p) = \text{first}(q)$ then $\text{split\_side}(p) = \text{right}$; otherwise $\text{last}(p) = \text{last}(q)$ and $\text{split\_side}(p) = \text{left}$.

In order to keep all these details together we enlarge the structure of each packet p:

$$p = \langle [\text{first}(p) \cdots \text{last}(p)], \text{node}(p), \text{level}(p), \text{active}(p), \text{remains}(p), \text{split\_side}(p) \rangle$$

We define predicates allowing us to differenciate the packets that point at a given node/level,

$$\begin{aligned}
\text{Left}(p) &\equiv \neg\text{remains}(p) \;\wedge \text{split_side}(p) = \text{left} \\
\text{Middle}(p) &\equiv \text{remains}(p) \\
\text{Right}(p) &\equiv \neg\text{remains}(p) \;\wedge \text{split_side}(p) = \text{right}
\end{aligned}$$

Finally we define the following sets, possibly empty, for each node/level (x, l) and stage t:

$$\begin{aligned}
\mathrm{L}_t(x,l) &= \{p \mid (\mathrm{node}_t(p), \mathrm{level}_t(p)) = (x,l) \ \wedge \mathrm{Left}_t(p)\} \\
\mathrm{M}_t(x,l) &= \{p \mid (node_t(p), level_t(p)) = (x,l) \ \wedge \mathrm{Middle}_t(p)\} \\
\mathrm{R}_t(x,l) &= \{p \mid (node_t(p), level_t(p)) = (x,l) \ \wedge \mathrm{Right}_t(p)\}
\end{aligned}$$

We define the *subskiplist* of a node/level (x, l), noted as $\mathrm{SS}(x, l)$. It is the skip list whose header is the node x and whose level is l. The node/levels that belong to the subskiplist $\mathrm{SS}(x, l)$ are those that can be reached from (x, l).

We define for each node/level (x, l) and stage t the following two sets of packets:

$$\begin{aligned}
\mathrm{D}_t(x,l) &= \{p \mid (\mathrm{node}_t(p), \mathrm{level}_t(p)) \in \mathrm{SS}(x, l-1)\} \\
\mathrm{F}_t(x,l) &= \{p \mid (\mathrm{node}_t(p), \mathrm{level}_t(p)) \in \mathrm{SS}(\mathrm{forward}(x,l), l)\}
\end{aligned}$$

We define for two packets p and q

$$\begin{aligned}
p \leq q &\equiv \mathrm{last}(p) \leq \mathrm{first}(q) \\
p \subset q &\equiv \mathrm{first}(q) \leq \mathrm{first}(p) \ \wedge \ last(p) \leq \mathrm{last}(q)
\end{aligned}$$

and for two sets of packets A and B we define

$$A \leq B \equiv \forall p, q : p \in [A] \ \wedge \ q \in [B] : p \leq q$$

Lemma 3.1. *At any stage $t > 0$ and any node/level (x, l)*

i) if $p \in \mathrm{L}_t \ \wedge \ \mathrm{M}_t \neq \emptyset \ \wedge \ \mathrm{L}_t \leq \mathrm{M}_t$ then $p \in \mathrm{D}_{t+1}$

ii) if $p \in \mathrm{R}_t \ \wedge \ \mathrm{M}_t \neq \emptyset \ \wedge \ \mathrm{M}_t \leq \mathrm{R}_t$ then $p \in \mathrm{F}_{t+1}$

iii) if $p \in \mathrm{L}_t \ \wedge \ \mathrm{D}_t \neq \emptyset \ \wedge \ \mathrm{L}_t \leq \mathrm{D}_t$ then $p \in \mathrm{D}_{t+1}$

iv) if $p \in \mathrm{R}_t \ \wedge \ \mathrm{F}_t \neq \emptyset \ \wedge \ \mathrm{F}_t \leq \mathrm{R}_t$ then $p \in \mathrm{F}_{t+1}$

Proof. $i)$: As $\mathrm{M}_t \neq \emptyset$ there exists a packet q such that $\mathrm{remains}_t(q)$. In a previous stage $t-1$ there has been a packet q' such that $q \subset q'$ and $\mathrm{first}(q') \leq \mathrm{key}(\mathrm{forward}(x, l)) < \mathrm{last}(q')$. As $\mathrm{L}_t \leq \mathrm{M}_t$ and $q \subset q'$ then $p \leq q$ and $p \leq q'$ because the set of packets are a partition of the original one ($p \cap q' = \emptyset$). Thus $\mathrm{last}(p) \leq \mathrm{first}(q') \leq \mathrm{key}(\mathrm{forward}(x, l))$ and p verifies $p \in \mathrm{D}_{t+1}$
$ii)$: It is proved as before.
$iii)$: As $\mathrm{D}_t \neq \emptyset$ there exists a packet q such that $\mathrm{last}(q) \leq \mathrm{key}(\mathrm{forward}(x, l))$. As for all packet $q \in \mathrm{D}_t(x, l) : p \leq q$ then $\mathrm{last}(p) \leq \mathrm{key}(\mathrm{forward}(x, l))$.
$iv)$: The proof is similar to that of $iii)$. ■

Lemma 3.2. *If for a given node/level (x, l) and stage t it holds:*

$$\#\mathrm{L} \leq 1 \ \wedge \ \#\mathrm{M} \leq 1 \ \wedge \ \#\mathrm{M} \leq 1 \ \wedge \ \mathrm{L} \leq \mathrm{M} \leq \mathrm{R}$$

then at most two packets can be moved from this node/level to any other. If both packets are sent to the same node/level they must have different split_side.

Proof. Suppose $\mathrm{M} \neq \emptyset$. There exists a packet p_m that is a subpacket of one packet p'_m that hits the forward key of (x, l) in the previous stage $t-1$; i.e. $\mathrm{first}(p'_m) \leq \mathrm{key}(\mathrm{forward}(x, l)) < \mathrm{last}(p'_m)$. Suppose that there exists another packet $p_l \in \mathrm{L}$; since $\mathrm{L} \leq \mathrm{M}$ and $p_l \leq p'_m$ it follows that $last(p_l) \leq \mathrm{key}(\mathrm{forward}(x, l))$ and then p_l will be sent to $(x, l-1)$ with $\mathrm{split\_side}(p_l) = \mathrm{left}$. By a similar reason, if R contains a packet p_r then it will be sent to $(\mathrm{forward}(x, l), l)$ with $\mathrm{split\_side}(p_r) = \mathrm{right}$. The packet p_m should be split into p_{mr} with split_side right and p_{ml} with split_side left, and the following relation holds $\mathrm{last}(p_{mr}) \leq \mathrm{key}(\mathrm{forward}(x, l))$ or $\mathrm{first}(p_{ml}) > \mathrm{key}(\mathrm{forward}(x, l))$. Then a packet generated by the split will be pushed either to the forward or down node/level, but in any case with a different split_side than the other packet that were in the L or R. If p_m does not split, it is at the left of $\mathrm{key}(\mathrm{forward}(x, l))$ with split_side right or at the right of the forward key with split_side left, i.e. $\mathrm{last}(p_m) \leq \mathrm{key}(\mathrm{forward}(x, l))$ or $\mathrm{first}(p_m) > \mathrm{key}(\mathrm{forward}(x, l))$. Then it has a different split_side than the other packet that goes to the same node/level. Therefore the lemma holds.
Supose that $\mathrm{M} = \emptyset$. At most there exist two packets $p_l \in \mathrm{L}$ and $p_r \in \mathrm{R}$. These can be moved to a different node/level or to the same, but in any case they have different split_side, by definition. ■

Lemma 3.3. *At any stage $t > 0$ and any node/level (x, l)*

i) $\#\mathrm{L}_t(x, l) \leq 1 \ \wedge \ \#\mathrm{M}_t(x, l) \leq 1 \ \wedge \ \#\mathrm{R}_t(x, l) \leq 1$.

ii) $\mathrm{L}_t(x, l) \leq \mathrm{D}_t(x, l) \leq \mathrm{M}_t(x, l) \leq \mathrm{F}_t(x, l) \leq \mathrm{R}_t(x, l)$.

Proof. We prove the the lemma by induction on the number of steps t.
Step $t = 0$. Initially p is located in $(\mathrm{header}(S), \mathrm{level}(S))$ with $\mathrm{split\_side}(p) = \mathrm{left}$ and $\neg\mathrm{remains}(p)$. Then $p \in \mathrm{L}_0(\mathrm{header}(S), \mathrm{level}(S))$ and the others sets are empty. Therefore, the lemma trivially holds.
Step $t + 1$.
Case $\mathrm{M}_t(x, l) \neq \emptyset$: it exists a packet $p_m \in \mathrm{M}_t(x, l)$, and two packets pp_l and pp_r that can arrive from the predecessor of (x, l) to (x, l).

- If $\#\mathrm{L}_t(x, l) = 1$ then there exists a packet $p_l \in \mathrm{L}_t(x, l)$. By Lemma 3.1 $p_l \in \mathrm{D}_{t+1}(x, l)$.
- If $\#\mathrm{R}_t(x, l) = 1$ then there exists a packet $p_r \in \mathrm{R}_t(x, l)$. By Lemma 3.1 $p_r \in \mathrm{F}_{t+1}(x, l)$.
- As pp_l comes from the predecessor and $\mathrm{split\_side}(pp_l) = \mathrm{left}$ then $pp_l \in \mathrm{L}_{t+1}(x, l)$.

- As pp_r comes from the predecessor with split_side(pp_r) = right then $pp_r \in \mathrm{R}_{t+1}(x,l)$.
- The packet p_m will be sent down or forward, or will be split into p_{ml} and p_{mr}. One of these would be sent down or forward, and at most one would remain in (x,l).

It is now clear, from the considerations above that

$$\#\mathrm{L}_{t+1}(x,l) \leq 1 \ \wedge \ \#\mathrm{M}_{t+1}(x,l) \leq 1 \ \wedge \ \#\mathrm{R}_{t+1}(x,l) \leq 1.$$

If pp_l is in $\mathrm{L}_t(pred(x,l))$, it is smaller than the packets in $\mathrm{D}_t(pred(x,l))$ and $\mathrm{F}_t(pred(x,l))$ by our inductive hypothesis *ii*), no matter that (x,l) is a forward or down node/level. If pp_l is in $\mathrm{M}_t(pred(x,l))$ then it is smaller than any packet in $\mathrm{F}_t(pred(x,l))$ only, but it should be forwarded and cannot be sent down. We can use a similar argument for pp_r and conclude that for all packets p

$$(\mathrm{node}_t(p), \mathrm{level}_t(p)) = (x,l) \Rightarrow pp_l \leq p \leq pp_r$$

Also, by inductive hypothesis *ii*) over (x,l) it holds for any p_l, p_m and p_r:

$$p_l \in \mathrm{L}_t(x,l), p_m \in \mathrm{M}_t(x,l), p_r \in \mathrm{R}_t(x,l) \Rightarrow p_l \leq p_m \leq p_r$$

- If pp_l comes from the predecessor then $\mathrm{L}_{t+1}(x,l) = \{pp_l\}$.
- $\mathrm{D}_{t+1}(x,l) = \mathrm{D}'_t(x,l) \cup \{p_l, p_{ml}\}$ if p_l exists and p_{ml} is sent down. The superscript *prime* means that $\mathrm{D}'_t(x,l) = \mathrm{D}_t(x,l)$ modulo splits.
- $\mathrm{M}_{t+1}(x,l)$ is $\{p_{ml}\}$, $\{p_{mr}\}$ or empty.
- $\mathrm{F}_{t+1}(x,l) = \mathrm{F}'_t(x,l) \cup \{p_r, p_{mr}\}$ if p_r exists and p_{mr} is sent down.
- If pp_r comes from the predecessor then $\mathrm{R}_{t+1}(x,l) = \{pp_r\}$.

Then on the node/level (x,l) at the following stage $t+1$ it holds

$$\mathrm{L}_{t+1}(x,l) \leq \mathrm{D}_{t+1}(x,l) \leq \mathrm{M}_{t+1}(x,l) \leq \mathrm{F}_{t+1}(x,l) \leq \mathrm{R}_{t+1}.$$

Case $\mathrm{M}_t(x,l) = \emptyset$: we consider two subcases; the first one is the case where $D_t(x,l) \neq \emptyset$ and the second one when this set is empty.
Case $\mathrm{D}_t(x,l) \neq \emptyset$:

- If $\#\mathrm{L}_t(x,l) = 1$ then the packet $p_l \in \mathrm{L}_t$ satisfies $p_l \in \mathrm{D}_{t+1}(x,l)$, by Lemma 3.1.
- If $\#\mathrm{R}_t(x,l) = 1$ and $\mathrm{F}_t(x,l) = \emptyset$, p_r can be sent down, forward or split. If $\mathrm{F}_t(x,l) \neq \emptyset$ the packet must be forwarded, by Lemma 3.1.
- If pp_l comes from the predecessor then $pp_l \in \mathrm{L}_{t+1}(x,l)$.

- If pp_r comes from the predecessor then $pp_r \in \mathrm{R}_{t+1}(x, l)$

Therefore, $i)$ holds at stage $t+1$.
If we write the inductive hypothesis $ii)$ for both the predecessor of (x, l) and (x, l) we get that, for all packets p,

$$(\mathrm{node}_t(p), \mathrm{level}_t(p)) = (x, l) \Rightarrow pp_l \leq p \leq pp_r.$$

and

$$p_l \in \mathrm{L}_t(x, l), p_d \in \mathrm{D}_t(x, l), p_r \in \mathrm{R}_t(x, l) \Rightarrow p_l \leq p_d \leq p_r$$

Hence $ii)$ also holds at stage $t+1$.

Case $\mathrm{D}_t(x, l) = \emptyset$: If $\mathrm{F}_t(x, l) \neq \emptyset$ we deal with the symmetrical case that the preceding one. Otherwise $\mathrm{F}_t(x, l) = \emptyset$. Writing the inductive hypothesis $ii)$ for the predecessor of (x, l), for any p

$$(\mathrm{node}_t(p), \mathrm{level}_t(p)) = (x, l) \Rightarrow pp_l \leq p \leq pp_r$$

and writing it for (x, l) it holds for any p_l and p_r

$$p_l \in \mathrm{L}_t(x, l), p_r \in \mathrm{R}_t(x, l) \Rightarrow p_l \leq p_r$$

- If pp_l and pp_r come from the predecessor, they will be the in $\mathrm{L}_{t+1}(x, l)$ and $\mathrm{R}_{t+1}(x, l)$, respectively.

- If p_l and p_m exist then both can move to $\mathrm{D}_{t+1}(x, l)$ or $\mathrm{F}_{t+1}(x, l)$, or the first can go down while the second is split or sent forward, or p_l is pushed down and p_r pushed forward, or the symmetrical cases. If only one of them exist, then it can go down, forward o be split.

Thus both $i)$ and $ii)$ hold at stage $t+1$.
■

Let us see now how to efficiently transform the search algorithm having at most three readers per node/level in each step into one with at most one reader per node/level. To do it, we need to stablish some protocol that guarantees that they can be distinguised, so each single step of the CREW algorithm can be decomposed in three distinct steps that perform the task in $O(1)$ time each and without read conflicts. This protocol will based upon on the functions remains and split_side.

Lemma 3.4. *The procedure route(p,P,S) can be implemented in the EREW model in time $O(1)$ with k processors.*

Proof. To do it let us consider the predicate remains and the function split_side in detail.

- When there are three packets q_1, q_2, q_3 only one of them, say q_i, satisfies remains(q_i), there is another one, say q_j, that satisfies $\neg$remains$(q_j) \wedge$ (split_side(q_j) = left) and the third one q_k verifies $\neg$remains$(q_k) \wedge$ (split_side(q_k) = right). A protocol such that first allows the packet q_i satisfying remains(q_i) to read, second allows the packet q_j with $\neg$remains$(q_j) \wedge$ (split_side(q_j) = left) to read and finally allows the packet q_k with $\neg$remains$(q_k) \wedge$ (split_side(q_k) = right) to read is exclusive.

- When there are two packets q_1, q_2 we have two cases. When there is a packet q_i with remains(q_i) the other one q_j verifies

$$\neg\text{remains}(q_j) \wedge \text{split\_side}(q_j) \in \{\text{left}, \text{right}\}$$

  If it holds $\neg$remains$(q_1) \wedge \neg$remains(q_2), then split_side$(q_1) \neq$ split_side(q_2). Therefore the preceding protocol also works.

■

Algorithm 4 give us this protocol explicitly.

```
procedure exclusive_read_forward_key(out b : key_type; in p : packet;
                                     in/out P : set_of_packets; in S : skip_list)

  if
      remains(p)  ⟶  b := key(forward(node(p), level(p))
  fi;
  if
      ¬ remains(p) ∧ splitside(p) = none  ⟶  b := key(forward(node(p), level(p))
  fi;
  if
      ¬ remains(p) ∧ splitside(p) = left  ⟶  b := key(forward(node(p), level(p))
  fi;
  if
      ¬ remains(p) ∧ splitside(p) = right  ⟶  b := key(forward(node(p), level(p))
  fi

end exclusive_read_forward_key
```

Algorithm 4: Exclusive read of keys

3.2 Efficiency of the search algorithm

We analyze in this subsection the number of stages that the search procedure makes before all packets have been routed to their final destinations. Our results for the expected number of stages and the result of the previous subsection on the time complexity of parallel routing will allow us to state Theorem 3.1 on the expected parallel time of the search algorithm.

Before we consider the expected number of stages that are done, we should note that in our presentation of the search algorithm (Algorithm 3) the presence or absence of active packets is checked 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 $\lceil \mathrm{L}(n) \rceil + \lceil \log_2 k \rceil$ stages, where $\mathrm{L}(n) = \log_{1/p} n$. 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 $\lceil \mathrm{L}(n) \rceil + \lceil \log_2 k \rceil$ stages is executed, etc (see Algorithm 5). As we shall soon prove in Lemma 3.5, 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 $O(\log n + \log k)$.

```
...
do  active(P) ⟶
    for j : 1..⌈L(n) + log₂ k⌉ do
        for all p : p ∈ active(P) do in parallel
            route(p, P, S)
        end
    end
od
...
```

Algorithm 5: Searching by runs.

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 $\mathrm{E}(X) \leq \mathrm{E}(Y)$. We will use $\mathrm{B}(n,\, p)$ 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 3.5. *Let $C_{n,k}$ be the random variable denoting the number of stages done by the search procedure before all packets become inactive, given k keys $a[1], \ldots, a[k]$ and a random skip list of size n. The expected value of $C_{n,k}$ is $O(\log n + \log k)$.*

Proof. Let C_i be 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. Clearly, $C_{n,k} = \max_{1 \leq i \leq k}\{C_i\}$. 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 [15] 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* [12] and *full horizontal steps* [15]). 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

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

The bound is given in the work by Pugh [18], although it is not presented in the same way as here. The results obtained by Devroye [4] 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, $E(C_{n,k}) \leq O(\log n) + E(W_n) + \log_2 k = O(\log n + \log k)$. ∎

From Lemmas 3.4 and 3.5 we have the next Theorem.

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 $O(n^{-2})$.*

The last claim in Theorem 3.1 can be proved using Chernoff bounds. Similar statements have been proved for the time complexity of the sequential search algorithm in skip lists and storage requirements [19]. Recall that the Chernoff tail bound lemma [3] states that for any random variable X, any value a and any $t > 0$

$$\Pr\{X \geq a\} \leq E(e^{tX})/e^{ta}. \tag{1}$$

The expectations $E(e^{tX})$ and $E(e^{-tX})$ for a binomial random variable $X = B(n,p)$ are

$$\begin{aligned} E(e^{tX}) &= \left(q + pe^{t}\right)^{n}, & \qquad (2)\\ E(e^{-tX}) &= \left(q + pe^{-t}\right)^{n}, \qquad q \equiv 1 - p & \qquad (3) \end{aligned}$$

And for a geometric random variable $X = NB(1,p)$

$$E(e^{tX}) = \left(\frac{q}{1 - pe^{t}}\right), \qquad q \equiv 1 - p. \tag{4}$$

Lemma 3.6. *Let $C_{n,k}$ be 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. Then, for all $\delta > 0$ and $0 < p < 1$, there exists a constant Δ_p such that*

$$\Pr\{C_{n,k} \geq \delta\, \mathrm{L}(n) + \lceil \log_2 k \rceil\} \leq \Delta_p / n^2.$$

In other words, the probability that the search time deviates significantly from the expected time decreases as $O(n^{-2})$.

Proof. Let $X_1 = NB(L(n) - 1, p)$, $X_2 = NB(1, 1-p)$, $X_3 = B(n, 1/np)$. Note that $C_{n,k} \leq_{\text{prob}} X_1 + X_2 + X_3 + L(n) + \lceil \log_2 k \rceil$.
Applying the Chernoff bound above (Eqs. 1 and 2) for X_3 and $t = (2/c_3)\ln(1/p)$ we have

$$\Pr\{X_3 \geq c_3 \cdot L(n)\} \leq n^{-2} \exp\left(p^{-1-2/c_3} - p^{-1}\right) = n^{-2}\,\gamma_p.$$

We shall remark that other carefully chosen values of t would yield a bound of the form n^{-c}. The same holds for the bounds that we derive next.
For X_2 and $t = (2/c_2)\ln(1/p)$ the bound is (Eqs. 1 and 4)

$$\Pr\{X_2 \geq c_2 \cdot L(n)\} \leq n^{-2}\left(\frac{1-p}{1-p^{1-2/c_2}}\right) = n^{-2}\,\beta_p.$$

Note that the bound is not well defined for $c_2 = 2$.
The third bound can be derived once we observe that $X_1 \geq z$ means that after z trials we have not yet seen $r = L(n) - 1$ successes. If we define Y_1 to be the number of successes in the z trials, then Y_1 follows a $B(z, p)$ distribution, and $X_1 \geq z$ implies $Y_1 \leq L(n) - 2$. Hence,

$$\Pr\{X_1 \geq z\} \leq \Pr\{Y_1 \leq L(n) - 2\}.$$

Using Eqs. 1 and 3,

$$\Pr\{X_1 \geq z\} \leq \Pr\{Y_1 \leq L(n) - 2\} \leq e^{t(\mathrm{L}(n)-2)}(1 + p(e^{-t} - 1))^z.$$

Taking $z = c_1\, L(n)$ and $t = 2\ln(1/p)$, for any $c_1 > 0$ and $n > 1$, we have:

$$\Pr\{X_1 \geq c_1\, L(n)\} \leq n^{-2}\, p^4.$$

Finally, we have

$$\Pr\{X_1 + X_2 + X_3 + L(n) + \lceil \log_2 k \rceil \geq \delta\, L(n) + \lceil \log_2 k \rceil\} \leq \frac{\Delta_p}{n^2},$$

where $\delta = c_1 + c_2 + c_3$ and $\Delta_p = p^4 + \gamma_p + \beta_p$. Since $C_{n,k} \leq_{\text{prob}} X_1 + X_2 + X_3 + L(n) + \lceil \log_2 k \rceil$ the lemma follows. ■

p	Δ_p	$n_p(0.25)$	$n_p(0.1)$	$n_p(0.01)$
0.20	40.84	13	21	64
0.25	11.61	7	11	35
0.30	5.57	5	8	24
0.35	3.49	4	6	19
0.40	2.56	4	6	16
0.45	2.06	3	5	15
0.50	1.78	3	5	14
0.55	1.61	3	5	13
0.60	1.51	3	4	13
0.65	1.46	3	4	13
0.70	1.45	3	4	13
0.75	1.46	3	4	13
0.80	1.51	3	4	13

Table 1: Values of Δ_p and n_p.

The value of Δ_p for several different values of p is given in Table 1. We have arbitrarily fixed the values of c_1, c_2 and c_3 to 0.1, 0.1 and 5.8. Therefore, we are computing Δ_p for $\delta = 6$.

The table also lists the values $n_p(0.25)$, $n_p(0.1)$ and $n_p(0.01)$, where $n_p(x)$ is the minimun n such that Δ_p/n^2 is less than x. It is interesting to note that the probability that the actual time of a search is six or more than six times larger than the expected time, i.e. we need to perform $\delta = 6$ or more runs, is less than 0.01, for $n \geq 70$ and a wide range of reasonable values of p ($0.2 < p < 0.8$). Figure 2 contains the plot of Δ_p/n^2 for $p = 0.25$, $p = 0.5$ and $p = 0.75$. We have marked the value $n = 35$ since $n_p(0.01) \leq 35$ for all of the three chosen values of p.

4 Insertion

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 6). First, the procedure *search_with_update* makes a search of the k keys using a slight variation of the parallel 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.

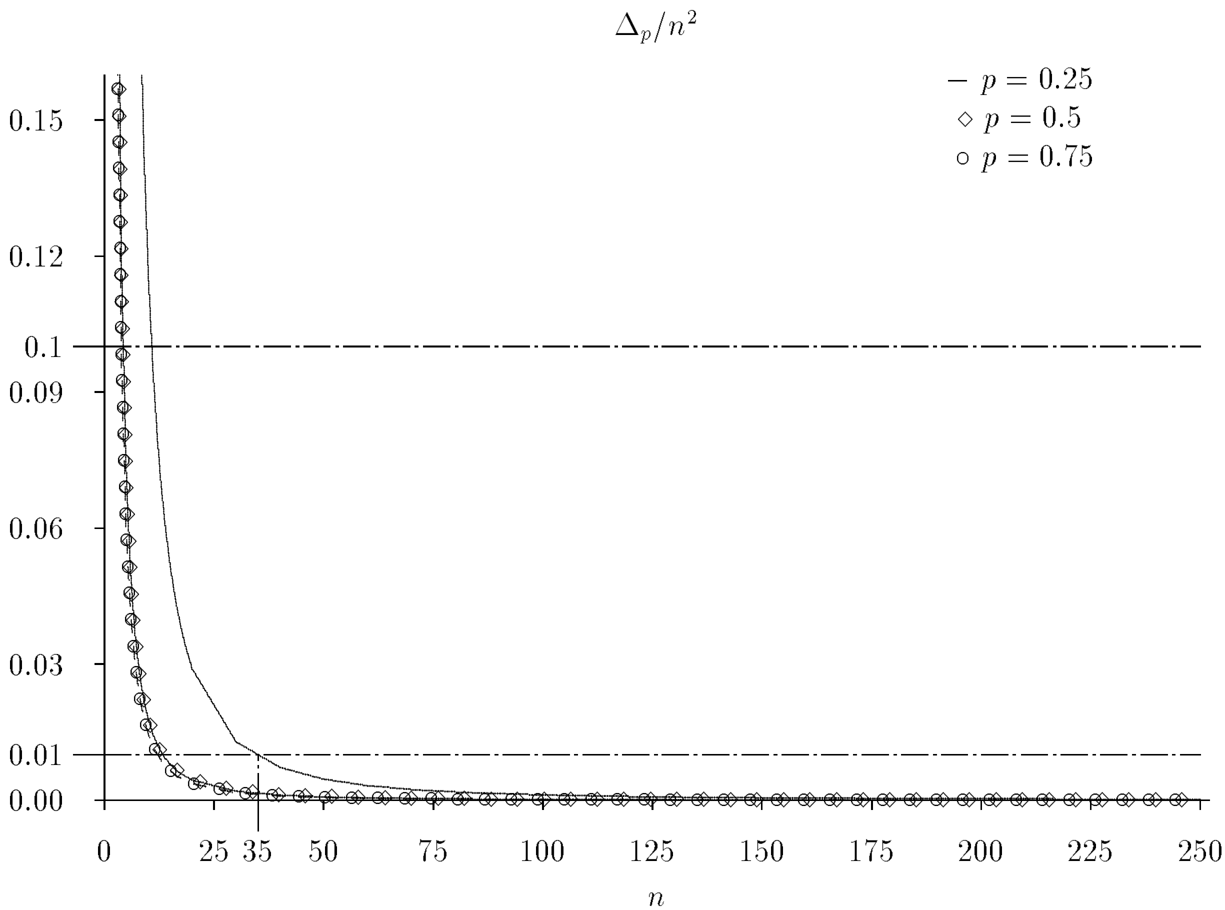

Figure 2: Plots of the bound Δ_p/n^2 for $p = 0.25$, 0.5 and 0.75

```
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+1;
         update_array = array [1..k, 1..M] of refnode_type;
         node_array   = array [1..k] 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 6: Insertion algorithm.

• Procedure *search_with_update*: Upon termination each key p has a reference to the nodes in S that are its *potential* predecessors for each level l. A node x is said to be *potential* predecessor at level l of the key a, if x would be the actual predecessor at level l of the node storing a, if a were the only key to be inserted and the node containing a had level ≥ 1. To know these potential predecessors, we must keep track of the path followed by the packets. Hence, for each packet p, we take an array *update*, 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:

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

• 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 geometric $NB(1, p)$ distribution, as in the sequential case. 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.

• 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 $m = \max\{level[i] : 1 \le i \le k\}$ useful rows and $k + 2$ columns. Any column i with $1 \le i \le k$ has $level[i]$ useful rows. Columns 0 and $k + 1$ are sentinels with m rows. The array *succ*, for $0 \le i \le k$, is defined as:

$$succ[i, l] = \text{Minimum } j \text{ to the right of } i, \text{ that is, } i < j \le k + 1, \text{ such that } level[j] \ge l.$$

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. It is clear that all the rows of *succ* can be computed independently. Also, it is not difficult to see that, in order to fill each of the rows, we need only easy tree computations.

• Procedure *merge*: This phase merges *succ* with S (see Algorithm 7). If $m > \mathrm{level}(S)$, 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*.

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 $i : 1 \leq i \leq k \wedge level[i] \geq l$ **do in parallel**
$j := succ[i, l]$;
if
$j = k + 1 \longrightarrow$ forward$(node[i], l) :=$ forward$(update[i, l], l)$
$\| \; j \neq k + 1 \wedge update[i, l] = update[j, l] \longrightarrow$ forward$(node[i], l) := node[j]$
$\| \; j \neq k + 1 \wedge update[i, l] \neq update[j, l] \longrightarrow$ forward$(node[i], l) :=$ forward$(update[i, l], l)$;
forward$(update[j, l], l) := node[j]$
fi
end;
$j := succ[0, l]$;
forward$(update[j, l], l) := node[j]$

end merge_level

var $l : 1..M$;
$i : 1..k$;

for all l : level$(S) \leq l \leq m$ **do in parallel**
for all $i : 1 \leq i \leq k$ **do in parallel**
$update[i, l] :=$ header(S)
end
end;
for $l : 1 \leq l \leq m$ **do**
merge_level$(succ, update, node, S, l)$
end;
level$(S) :=$ max(level$(S), m)$

end merge

Algorithm 7: Procedures merge and 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 7). 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.

4.1 Efficiency of the insertion algorithm

The search for the places to insert the k new items needs parallel expected time $O(\log n + \log k)$ using $O(k)$ processors (Theorem 3.1). Gathering the information in the update arrays during the search does only introduce an additional constant cost for each routing step. But the information collected in these update arrays must be spread to all the processors (now one processor is associated to every key). 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 $O(\text{level}(S) + \log k)$ using k processors. Since the expected level of S is $O(\log n)$ the expected time to spread the update arrays is $O(\log n + \log k)$ using $O(k)$ processors.

The next step in the insertion procedure requires the computation of a random level for each new item to be inserted. The expected time to generate the k random levels is equivalent to the expected height of a skip list of size k. Therefore, after an expected number $O(\log k)$ of iterations, the level of each key has been independently assigned.

Then, k new nodes must be allocated to store the new items. The parallel dynamic memory manager needs $O(\log k)$ steps to allocate the k new nodes [11, 17].

To create the array *succ* with the information about possible successors of the new items, we need first to compute the maximum level m among the levels of the new items. This can be obviously done in $O(\log k)$ steps. Using k/m processors for each row, it takes $O(m)$ extra time 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.

Finally, a call to the procedure *merge* is performed. The execution of this procedure with $O(k)$ processors takes parallel expected time $O(\log k)$, since *merge_level* has cost $O(1)$ using $O(k)$ processors and *merge* calls *merge_level* m times, once for each level.

Taking all these contributions into account Theorem 4.1 follows.

Theorem 4.1. *In a skip list, the insertion 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 inserted and n is the length of the skip list. Moreover, the probability that the performance of the insertion algorithm deviates from the expected time decreases as $O(n^{-2} + k^{-2})$.*

The small probability of bad performance for the insertion algorithm can be proved using Chernoff bounds, following similar arguments to those used in Subsection 3.2.

5 Deletion

We consider 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 *succ*, *pred* and *last* giving informations about sucessors, predecessors and blocks of consecutive nodes to be deleted. Now, for each node to be removed and each of its levels, we must know whether its predecessors will remain or be removed. Which one of these cases holds can be checked in parallel using the *update* and the three preceding skip arrays. Both cases can be managed with simple parallel address arithmetic techniques similar to those used in the insertion algorithm.

- Procedure *search_with_update*: Identical to the one given for the insertion algorithm.
- Procedure *make_succ*: It constructs the array

$$succ[i,l] = \text{minimum } j \text{ to the right of } i \ (i < j \leq k+1), \text{ such that } level[j] \geq l.$$

It is similar to the skip array *succ* used in the insertion procedure, but now the number of rows in *succ* is computed from the actual levels of the nodes to be deleted.

- Procedure *make_pred*: This procedure builds an skip_array called *pred*, that provides information on the predecessor of the items to be deleted.

$$pred[i,l] = \text{maximum } j \text{ to the left of } i \ (0 \leq j < i), \text{ such that } level[j] \geq l.$$

- Procedure *make_last*: Consider a node $node[i]$ to be deleted. At level l this node heads the linked list

$$node[i], \mathit{forward}(node[i],l), \mathit{forward}(\mathit{forward}(node[i],l),l), \ldots$$

Maybe a block of consecutive nodes starting from $node[i]$ will be deleted too. In order to chain the remaining nodes, we need to know the index of the rightmost element in this block. We call it $last[i,l]$. Therefore the nodes

$$node[i], \mathit{forward}(node[i],l), \mathit{forward}(\mathit{forward}(node[i],l),l), \ldots, node[last[i]]$$

will be deleted, but $\mathit{forward}(node[last[i,l],l)$ remains in S. Formally,

$$\begin{aligned} last[i,l] \quad = \quad & \text{Minimum index } j : i \leq j \leq k, \text{ of a node } node[j] \text{ in } S \text{ to} \\ & \text{ be deleted such that his forward node at level } l, \text{ i.e.} \\ & \mathit{forward}(node[j],l), \text{ will remain in } S. \end{aligned}$$

There are two cases in the computation of $last[i,l]$. When $succ[i,l] = k+1$ we get the right sentinel and the forward pointer of $node[i]$ at level l points to a node remaining in S. Therefore $last[i,l] = i$. When $succ[i,l] \neq k+1$ we have

$$\text{“}\mathit{forward}(node[j],l) \text{ remains in S”} \equiv \text{“}\mathit{forward}(node[j],l) \neq node[succ[j,l]]\text{)”}$$

and the element $last[i,l]$ is given by

$$last[i,l] = min(j :\ i \leq j \leq k\ \wedge\ level[j] \geq l\ \wedge\ \text{forward}(node[j], l) \neq node[succ[j,l]])$$

The computation of the array $last[i,j]$ has expected cost $O(\log k)$ with $O(k)$ processors.

• Procedure *remove*: It removes the elements to be deleted. Given a $node[i]$ to be deleted and a level l the preceding node in S at this level is $update[i,l]$. There are two cases. Fhe first one happens when $update[i,l]$ needs to be removed. This can be tested easily because

$$\text{“}update[i,l] \text{ will be deleted”} \equiv \text{“}pred[i,l] \neq 0 \wedge update[i,l] = node[pred[i,l]]\text{”}$$

As $node[pred[i,l]]$ will be deleted too the pointer $forward(node(pred[i,l]), l)$ is redundant and the algorithm does not take care of it.
The second one appears when $update[i,l]$ remains in S.

$$\text{“}update[i,l] \text{ remains”} \equiv \text{“}pred[i,l] = 0 \vee pred[i,l] \neq update[i,l]\text{”}$$

The node $update[i,l]$ remains in the list and its forward pointer at level l needs to be updated with $forward(node[last[i,l]], l)$.

```
procedure delete (in a : array [1..k] of key_type; in/out S : skip_list);

    procedure search_with_update(...)

    procedure make_pred(...)
    procedure make_succ(...)

    procedure make_last(in level : level_array; in m : 1..M; in node : node_array; in S : skip_list;
                        in succ : skip_array; out last : skip_array)
        var i : 0..k; l : 1..M;

        for all l : 1 ≤ l ≤ m do in parallel
            for all i : 1 ≤ i ≤ k ∧ level[i] ≥ l do in parallel
                if
                     succ[i, l] = k + 1  ⟶  last[i, l] := i
                  [] succ[i, l] ≠ k + 1  ⟶  last[i, l] := min(j : i ≤ j ≤ k ∧
                                                              level[j] ≥ l ∧
                                                              forward(node[j], l) ≠ node[succ[j, l]])
                fi
            end
        end
    end make_last

    procedure remove(...)

    var pred, succ, last : skip_array;
        update : update_array;
        node : node_array;
        level : level_array;
        m : 1..M;
        i : 1..k;

    search_with_update(a, S, node, update);
    for all i : 1 ≤ i ≤ k do in parallel
        level[i] := level(node[i])
    end;
    m := max(level[i] : 1 ≤ i ≤ k);
    make_pred(level, m, pred);
    make_succ(level, m, succ);
    make_last(level, m, node, S, succ, last);
    remove(pred, last, update, node, m, S)
end delete
```

Algorithm 8: Deletion algorithm.

```
procedure remove (in pred, last : skip_array; in update : update_array;
                  in node : node_array; in m : 1..M; in/out S : skip_list)

    procedure remove_level (in pred, last : 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 ∧ l ≤ level[i] do in parallel
            j := pred[i, l];
            if
                j = 0 cor update[i, l] ≠ node[j] ⟶
                        forward(update[i, l], l) := forward(node[last[i, l]], l)
              ▯ j ≠ 0 cand update[i, l] = node[j] ⟶ skip
            fi
        end;
        for all i : 1 ≤ i ≤ k do in parallel
            free(node[i])
        end
    end remove_level

    var l : 1..M;

    for l : 1 ≤ l ≤ m do
        remove_level(pred, last, update, node, S, l)
    end;
    do  forward(header(S), level(S)) = NIL ⟶
                level(S) := level(S) − 1
    od

end remove
```

Algorithm 9: Procedures remove and remove_level.

Theorem 5.1. *In a skip list, the deletion 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 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 $O(n^{-2} + k^{-2})$.*

6 Implementation issues

The high level abstraction of packets and the routing of these packets along of the skip lists is useful to explain the search algorithm and to convince oneself and others of the correctness of the algorithm, but it seems to be far from the capabilities of most current parallel languages. Several examples will be given in the C*® language for the data parallel Connection Machine® [20].

A simple way to implement the search algorithm (Algorithm 3) in a SIMD machine is to define a parallel arrays of size k, so each processor holds a key, an active/inactive status, etc. For instance, we could dynamically declare a *shape* of size k and then a parallel variable of this shape in C*, where each component of the parallel variable is a record with several fields. One of the fields, `key`, holds the keys to be searched. The other fields are: a field of boolean values `active` , two fields of integers `first` and `last`, a field of references to nodes in the skip list `node`, a field of integers `level`, and three additional fields: `curr_key`, `remains`, and `split_side`, to avoid concurrent read conflicts (see subsection 3.1). We can think of each processor containing an *slice* associated with a particular key: the i-th proccessor contains the i-th key and the i-th value of each of the other fields.

The fact that a packet p holds the keys $a[i], \ldots, a[j]$ (i.e. $\mathrm{first}(p) = i$ and $\mathrm{last}(p) = j$) at some stage of the algorithm is represented by storing these boundary pointers in the i-th components of the fields `first` and `last`: $[i]$`T.first` $= i$ and $[i]$`T.last` $= j$. If p is an active packet, then the value of the i-th component of the `active` field is 1 (true), whereas the rest of the components in the `active` field that are associated to the other keys in the same packet p are set to 0 (false). That means that the processors associated with the first key of each of the active packets are the only ones that perform useful work. These processors will be called *leaders*.

We know give the C* code for the setup of the searching procedure:

```
shape []search_shape;   /* a variable of shape search_shape is a
                          1-dimensional array, where each of the
                          components is held by a processor */
                        /* <type>:search_shape is the type of a parallel
                          variable of shape 'search_shape' where each
                          component/element is of type <type> */
#define LEFT 0
#define RIGHT 1

typedef struct {
  key_type key;
  int first, last;
  int level;
  refnode_type node;
  bool active;
  key_type curr_key;
  bool remains;
  int split_side; } search_elem;

typedef search_elem:search_shape search_array;

void search(search_array *a, int k, skip_list S)
{
```

```
/* initially, the 'key' field stores the k keys; the result
   is returned in the 'node' field */

[0](a -> first) = 0; [0](a -> last) = k - 1;
[0](a -> node) = header(S); [0](a -> level) = level(S);

[0](a -> split_side) = LEFT; /* this is arbitrary */

/* only processor nr. 0 is active at the beginning */
with (search_shape) {
 a -> active = (pcoord(0) > 0);
 a -> remains = 0;
 ...
```

The first lines of code of the *search* procedure are the equivalent ones to

$$p := < [1..k], header(S), level(S), true >;$$
$$P := \{\ p\ \};$$

The test for active packets can be accomplished by performing a parallel OR of all components of the active field. The inner **for all** loop of our search algorithm has to be applied only to those packets that are still active. These means that we should select only those processors where the `active` field is true.

The search procedure can be then written as follows:

```
void search(...)
{
  int i, run_size;

  /* initializations */ ...

  with (search_shape)
    while (|= a -> active)
      for (i = 0; i < run_size; i++)
        where (a -> active)
          route(a, S);

  /* spread the node of a packet to its keys, for all packets */ ...
}
```

The implementation of the procedure *route* is also very easy, although three separate reading phases are needed. Let us now see how the procedure *route* could be implemented in C* if we assume that all the *leaders* have in `curr_key` the key of the successor at level `level` of the current `node`.

```
void route(search_array *a, skip_list *S)
{
  int:search_shape status;

  /* search_shape is the current shape; a -> active
     is the current context */

  /* here goes the code to get the current key in exclusive-read
     fashion using the remain and split_side fields */

  status = NULL;
  compute_status(a, &status);
  push_forward(a, S, status);
  push_down(a, S, status);
  split_and_push(a, S, status);
}
```

After the call to the procedure *compute_status* each active processador (leader of a packet) has an indication on what should it do next with its associated packet.

```
void compute_status(search_array *a, int:search_shape *st)
{
/* search_shape is the current shape; the current context
   is a -> active or nested inside a -> active */


  where (a -> curr_key < [a -> first](a -> key))
     *st = FORWARD;

  where (*st == NULL)
     where (a -> curr_key >= [a -> last](a -> key))
       *st = DOWN;
     else
       *st = SPLIT;
}
```

The other procedures only apply to the active processors where the `status` flag has the appropriate value. We shall remark that if a packet is split into two subpackets then both are made active and point to the node/level where the original packet was split, but not routed. If the routing of the subpackets had to be done within the same stage where the subpackets were created, the C* code would get more involved or use unconditional parallel operations. The reason is that, although the processors associated to the subpackets generated by a `split_and_push` operation are active, only one of them is actually selected by the current context. In our implementation, the routing of the subpackets created by `split_and_push` is delayed until the next stage, where one or both of the subpackets will move forward or down. This modification to the algorithm in Section 3 clearly allows up to four packets pointing to the same node/level and requires a slight variation of the code to read the current keys in a exclusive basis.

```
void push_forward(search_array *a, skip_list S, int:search_shape status)
{
  where (status == FORWARD) {
     /* the node (a -> node) of each "forwarding" packet changes
        to its successor at the current level (a -> level);
        this heavily depends on how the skip list
        is represented ... */
     a -> remains = 0;
    }
}

void push_down(search_array *a, skip_list S, int:search_shape status)
{
  where (status == DOWN) {
     /* depending on the representation of S, something should
        be done here ... */
      a -> level--;
      a -> active = (a -> level > 0);
      a -> remains = 0;
    }
}

void split_and_push(search_array *a, skip_list S, int:search_shape status)
{
  int:search_shape m;

  where (status == SPLIT) {
      m = (a -> first + a -> last) / 2;
      [m + 1](a -> last) = (a -> last);  a -> last = m;
      m++;
      [m](a -> first) = m;
      [m](a -> level) = a -> level; [m](a -> node) = a -> node;
      [m](a -> active) = 1;
      a -> remains = true; [m](a -> remains) = true;
      a -> split_side = RIGHT; [m](a -> split_side) = LEFT;
```

7 Conclusions

We have presented an EREW implementation of a parallel dictionary based on a skip list having two main points:

i) It is easy to explain and justify and does not use recursion. Therefore it seems possible to program it becoming not only a theoretical tool but a practical one.

ii) Moreover it needs few processors and has good expected behaviour.

Skip lists share some of the properties of trees whilst provide the simplicity of the linked lists. The main ideas involved in the search algorithm can be schematized as: “route a set of packets on the skip list and split a packet whenever it collides with some key”. This algorithm reinforces the vision of skip lists as trees. The approach taken the insertion algorithm can rephrased as: “after the search phase has been done it works as a parallelization of the usual insertion algorithm on sequential linked lists using an extra array to do fast address arithmetic”. The same can be said about the deletion algorithm.

Those algorithms work on the linked lists that form the skip list use havily the behaviour of skip lists as a collection of linked lists.

We think skip lists could become a practical and useful data structure for SIMD machines.

Acknowledgements

We thank Torben Hagerup to help us with this research.

References

[1] M. J. Atallah and S. Rao Kosaraju. A generalized dictionary machine for VLSI. *IEEE Trans. on Comp.*, C-34(2):151–155, February 1985.

[2] H. Bast, M. Dietzfelbinger, and T. Hagerup. A perfect parallel dictionary. In I. M. Havel and V. Koubek, editors, *Proc. 17th Mathematical Foundations of Computer Science, LNCS 598*, pages 133–141. Springer-Verlag, 1992.

[3] H. Chernoff. A measure of asymptotic efficiency for tests of hypothesis based on the sum of observations. *Ann. Math. Statist.*, 23:493–507, 1952.

[4] L. Devroye. A limit theory for random skip lists. *The Annals of Applied Probability*, 2(3):597–609, 1992.

[5] M. Dietzfelbinger and F. Meyer auf der Heide. Dynamic hashing in real time. In J. Buchmann, H. Ganzinger, and J. W. Paul, editors, *Informatik. Festschrift zum 60. Geburtstag von Günter Hotz. Teubner-Texte zur Informatik 1*, pages 95–119. B. G. Teubner, 1992. Also in Proc. 17th ICALP, LNCS 443, 1990.

[6] M. Dietzfelbinger and F. Meyer auf der Heide. An optimal parallel dictionary. *Information and Control*, 102(2):196–217, February 1993.

[7] T. Duboux, A. Ferreira, and M. Gastaldo. MIMD dictionary machines: from theory to practice. In L. Bougé, M. Cosnard, Y. Robert, and D. Trystram, editors, *Proc. of the 2nd Joint Int. Conf. on Vector and Parallel Processing (CONPAR92-VAPPV), LNCS 634*, pages 545–550. Springer-Verlag, 1992.

[8] J. Gabarró and R. Gavaldà. An approach to correctness of data parallel algorithms. *J. Parallel and Distributed Computing*, 1993. To be published.

[9] M. Gastaldo. Dictionary machine on SIMD archictectures. Technical Report 93–19, Laboratoire de l'Informatique du Parallélisme (LIP), July 1993.

[10] J. Gil, Y. Matias, and U. Vishkin. Towards a theory of nearly constant time parallel algorithms. In *Proc. 32nd Foundations of Computer Science*, pages 698–710, 1991.

[11] W. D. Hillis and G. L. Steele, Jr. Data parallel algorithms. *Comm. ACM*, 29(12):1170–1183, December 1986.

[12] P. Kirschenhofer and H. Prodinger. The path length of random skip lists. *Acta Informatica*, 1994. To appear.

[13] C. E. Leiserson. *Area-Efficient VLSI Computation.* PhD thesis, Carnegie-Mellon University, 1981. ACM Doctoral Dissertation Award 1982. The MIT Press, 1983.

[14] L. Mounier and G. Utard. Sémantique axiomatique des langages à parallélisme de données; automatisation de la preuve de programmes. Technical Report 93–08, Laboratoire de l'Informatique du Parallélisme (LIP), March 1993.

[15] T. Papadakis, J. I. Munro, and P. V. Poblete. Analysis of the expected search cost in skip lists. In J. R. Gilbert and R. Karlsson, editors, *Proc. 2nd Scandinavian Workshop on Algorithm Theory, LNCS 447*, pages 160–172. Springer-Verlag, 1990.

[16] W. Paul, U. Vishkin, and H. Wagener. Parallel computation on 2–3 trees. In J. Díaz, editor, *Proc. 10th International Colloquium on Automata, Programming and Languages, LNCS 154*, pages 597–609. Springer-Verlag, July 1983. Also in RAIRO Informatique Théorique, pages 398–404, 1983.

[17] W. Paul, U. Vishkin, and H. Wagener. Parallel computation on 2–3 trees. *RAIRO Informatique Théorique*, pages 398–404, 1983.

[18] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Comm. ACM*, 33(6):668–676, 1990.

[19] S. Sen. Some observations on skip-lists. *Inform. Proc. Lett.*, 39(3):173–176, 1991.

[20] Thinking Machines Corporation, Cambridge, MA. C* *Programming Guide. Version 6.0.2*, June 1991.