ELSEVIER

Theoretical Computer Science 158 (1996) 1–33

Theoretical Computer Science

A design of a parallel dictionary using skip lists☆

Joaquim Gabarró, Conrado Martínez*, Xavier Messeguer

Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, Pau Gargallo 5, 08028-Barcelona, Spain

Received April 1994; revised November 1994
Communicated by M. Nivat

Abstract

We present a top–down design of a parallel PRAM dictionary using skip lists. More precisely, we give detailed algorithms to search for, insert or delete k elements in a skip list of n elements in parallel. The algorithms are iterative and easy to implement on real machines. We discuss some implementation issues and give concrete examples in C^*. The algorithms run on an EREW PRAM in expected time $O(\log n + \log k)$ using k processors. We also show an explicit protocol to avoid read conflicts thus obtaining an efficient EREW version of our algorithms.

Although the asymptotic performance of the parallel skip list algorithms is not better compared to that of other parallel dictionaries, they are a practical alternative. Skip list algorithms are very simple and there is a small probability of large deviations from their expected performance.

1. Introduction

Parallel dictionaries have been widely studied in the recent years. In a systolic framework, priority queues and search trees algorithms were designed by Leiserson [18]. Later, Atallah and Kosaraju [1] developed a generalized dictionary where a sequence of operations can be pipelined at a constant rate. Paul et al. [22, 23] have proposed efficient PRAM algorithms to dynamically maintain a parallel dictionary on 2–3 trees working on "batches" of k keys simultaneously. They have considered an EREW PRAM machine with k processors. Parallel search, insertion and deletion algorithms for k items in a 2–3 tree storing n items were shown to take time $O(\log n + \log k)$ in the worst-case. Both the insertion and deletion are rather

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

* Corresponding author. E-mail: conrado@lsi.upc.es.

SSDI 0304-3975(94)00288-6

sophisticated. More recently, Higham and Schenk [14] have studied parallel algorithms for the dynamic maintenance of a dictionary on B-trees that exhibit a performance comparable to that of the algorithms for 2–3 trees. Also, it is possible to design a parallel dictionary using hashing. Some references in this active research area are [2, 7, 8, 13].

Parallel dictionaries have been actually implemented on massively parallel machines. Duboux et al. [9] have implemented a MIMD dictionary in a Volvox IS860 with 8 nodes using sequential algorithms on 2–3–4 trees as local data structures. More recently, Gastaldo [12] has implemented a parallel dictionary in a SIMD machine, the MasPar MP-1. A linear array is used to represent local dictionaries in each processor. Assuming 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.

We add skip lists to the "general picture" of parallel algorithms by developing algorithms for parallel search and update of these data structures; the framework of our study is the same as in the previous works by Paul et al. for 2–3 trees and by Higham and Schenk for B-trees [14]. A preliminary version of the present work has appeared in [11]. A skip list is a randomized data structure used to represent abstract data types such as dictionaries and ordered lists. Skip lists were introduced by Pugh in 1990 [25] as 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 the algorithms. Some of the main aspects of this work are:

- The proposed algorithms extensively rely on two basic design ideas. First, the routing of a set of packets along the skip list. Second, the extensive use of address arithmetic to reconstruct the data structure, for both insertions and deletions.
- Special care has been taken to obtain a clear, precise and workable description of the algorithms. To get them we have followed a top–down approach. We stop our refinements in a point such that it will be easy to implement the algorithms on real parallel languages.
- Practical considerations are taken into account. In particular, we discuss some aspects of a C* implementation of the algorithms.
- To get accurate performance bounds in the EREW model we need to do a detailed analysis of the read conflicts. We characterize in very precise way the flow of packets during routing phases and give an explicit protocol to avoid read conflicts in constant time.

The paper is organized as follows: in Section 2 we review some basic facts about skip lists. In Section 3, we present a top–down design of the algorithm to search for k keys in parallel. To derive it, we use the skip list as a tree. Assuming that we are given an ordered array of k keys, the algorithm routes a set of packets containing the keys

along the skip list, until the packets stop at appropriate places. We apply stepwise development techniques along the lines suggested in the work by Gabarró and Gavaldà [10] and Bougé et al. [4]. 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 and B-trees [14, 22, 23]. Our parallel search algorithm has expected performance $O(\log n + \log k)$, where k is the number of keys to be searched for and n is the size of the skip list, and can be executed in an EREW PRAM machine using k processors. In Sections 4 and 5 we obtain the algorithms for insertion and deletion. Both algorithms deal with 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 to do parallel address arithmetic. Both algorithms are iterative. Their expected time is $O(\log n + \log k)$, as in the case of the parallel search. We also discuss several implementation-related issues and give some hints for the implementation of our algorithms in Section 6. Finally, Section 7 is devoted to conclusions.

2. Skip lists

Skip lists are randomized data structures introduced by Pugh in 1990 [25]. 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 [27]. They have a rich and interesting probabilistic analysis; consider, for instance [6, 16, 17, 20, 21].

We shall assume that the items to be stored in a skip list S (from now on, S will always denote a skip list) have different keys drawn from some totally ordered set. A nonempty skip list consists of several nonempty sorted linked lists. All the items are stored in the list of *level* 1. Some of the them also belong to the list of level 2, and so forth (see Fig. 1). Each item x in S has a key denoted as $\mathrm{key}(x)$ and a positive integer $\mathrm{level}(x)$. If $\mathrm{level}(x) = l$, it means that x belongs to the linked lists of level $1, 2, \ldots, l$. The levels of the items are given by independent geometrically distributed random variables with parameter $1 - p$. Therefore, the probability that an item x has level l is

$$\Pr\{\mathrm{level}(x) = l\} = p^{l-1}(1-p), \quad l \geqslant 1.$$

To implement a skip list, we need to allocate a node for each item. Each node x contains the item and $\mathrm{level}(x)$ pointers. The successor of x at level l, denoted $\mathrm{forward}(x, l)$, is given by the lth *forward* pointer of x. A header node, $\mathrm{header}(S)$, points to the first node of each linked list. We write $\mathrm{level}(S)$ to denote the maximum level

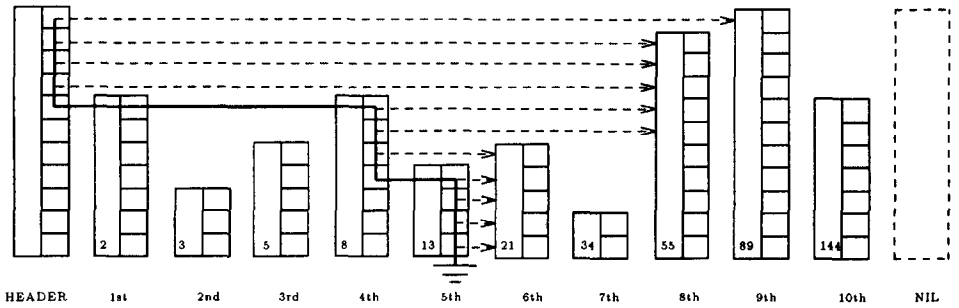

Fig. 1. A skip list of 10 items and the search path to the interval (13, 21].

among the levels of its items. The level of S is also called its *height*. We will use the following conventions:

- 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 and **NIL** is given a key greater than any legal key: key(**NIL**) = $+\infty$;
- The level of header(S) is level(header(S)) = level(S) and it stores a dummy key smaller than any legal key: key(header(S)) = $-\infty$;
- Each node also stores as many copies of its key as its level indicates. This convention simplifies the analysis of read conflicts (see Section 3.1), although it can be avoided (see Section 6).

Let $x \neq$ **NIL** be an item of S and l be some integer such that $0 \leqslant l \leqslant \text{level}(x)$, we write

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

For instance, in Fig. 1, wall(header(S), 7) is the 8th item in S and its key in 55. Note that wall(x, 0) = forward(x, l). Let $I(x, l)$ be the interval:

$$I(x, l) = (\text{key}(x), \text{key}(\text{wall}(x, l))] = \{k \mid \text{key}(x) < k \leqslant \text{key}(\text{wall}(x, l))\}.$$

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

Lemma 2.1. *Let S be a skip list and let a be a key such that $a \in I(x, l)$, for $x \neq$ **NIL** and $1 \leqslant l \leqslant \text{level}(x)$.*

Let b be the key stored at the successor of x at level l, that is, $b = \text{key}(\text{forward}(x, l))$. Then,

- *If $a > b$ then $a \in I(\text{forward}(x, l), l)$ and $\text{level}(\text{forward}(x, l)) = l$.*
- *If $a \leqslant b$ then $a \in I(x, l-1)$.*

Moreover, if $a \in I(x, 0)$ then $a \in S \equiv (a = \text{key}(\text{forward}(x, 1)))$.

In the following we sketch the sequential search and update algorithms [25].

(1) *Sequential search algorithm*: Given S and a key a, the search procedure returns the unique *node* in S such that $\text{key}(node) < a \leqslant \text{key}(\text{forward}(node, 1))$. It works moving the key a forward or down through S until it reaches *node*. In any given stage the key is said to be at a node/level (x, l), called the *current node/level*. Initially the current node/level is set to (header(S), level(S)). The search procedure iterates maintaining the invariant $a \in I(x, l)$ until the current level l is 0. In each iteration the current node/level is changed according to Lemma 2.1. It is useful to collect the key, the current node/level and the loop condition in a *packet* p

$$p \equiv \langle a, x, l, (l > 0)\rangle \equiv \langle \text{key}(p), \text{node}(p), \text{level}(p), \text{active}(p)\rangle,$$

and rephrase the search algorithm in terms of a packet moving through the skip list, until it becomes inactive.

(2) *Sequential insertion algorithm*: Assume, w.l.o.g., that the key a to be inserted does not belong to S. The insertion has three main phases. First, we search for a to locate the insertion point for the new item. It is also necessary to collect information about the would-be predecessors of the new item in each list. This can be done by means of an array called *update* (see [25]) such that

$update[l]$ = "the unique node in S such that

$$\text{key}(update[l]) < a \leqslant \text{key}(\text{forward}(update[l], l))", \quad 1 \leqslant l \leqslant \text{level}(S).$$

As $update[l]$ is the last visited node at level l during the search for key a, it is easy to modify the standard search procedure to compute also the *update* array as defined above. In the second phase a random level is chosen and a new node is allocated for the new item. Finally, the third phase modifies the necessary links to add the new node, using the *update* array.

(3) *Sequential deletion algorithm*: It goes along the same lines as the insertion algorithm. After the search for the item to be deleted, the links of the predecessors (given by the *update* array) are changed to remove the item.

3. Search

The algorithm searches for k keys by routing packets along the skip list S. A similar approach was developed for 2–3 trees by Paul et al. [22, 23] and by Higham and Schenk [14]. To be precise, given a skip list S of n items and an ordered array $a[1..k]$ with k keys, the search algorithm returns an array node$[1..k]$ such that:

node$[i]$ = a pointer to the unique node x in S such that

$$\text{key}(x) < a[i] \leqslant \text{key}(\text{forward}(x, 1)), \quad 1 \leqslant i \leqslant k.$$

Fundamental to our search algorithm is the notion of *packet*. A packet p stores a current node/level, an active status, and two indexes i and j, $1 \leqslant i \leqslant j \leqslant k$, representing the subarray $a[i..j]$ of $a[1..k]$. We write $\text{first}(p) = i$ and $\text{last}(p) = j$.

Formally,

$$p \equiv \langle [\text{first}(p)..\text{last}(p)], \text{node}(p), \text{level}(p), \text{active}(p)\rangle.$$

We denote by P the set of all the packets that exist in any given stage of the search. This set induces a full partition of the array $a[1..k]$, i.e. each key is "held" by one and only one packet. This property is maintained along the search. A packet p is said to be active if its level is not null: $\text{active}(p) \equiv (\text{level}(p) > 0)$. Inactive packets are those that have reached their final "destination". The subset of active packets is denoted by $\text{active}(P)$.

At the very start of the algorithm (see Algorithm 1), an active packet containing all k keys is "injected" into the skip list S by making (header(S), level(S)) its current node/level. In each stage, each active packet is routed by moving it forward or down, or it is split into two packets and one of these is moved forward or down. The main loop of the search ends when all the packets become inactive. At most k processors are needed to execute this loop; we need one processor to route in parallel each active packet. After the main loop, the array *node* is filled by spreading the current node of each packet to its keys (see Algorithm 1). We now describe each procedure used by the search algorithm in more detail.

(1) Procedure *route*: At each stage, all active packets are routed through S. The procedure *route*(p, P) moves or splits p in order to maintain the invariant of the main loop:

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

Given a packet p whose current node/level is (x, l), the procedure *route* compares $a[\text{first}(p)]$ and $a[\text{last}(p)]$ with $b = \text{key}(\text{forward}(x, l))$. Then it pushes forward, down or splits p, depending on the outcome of the comparisons, using the procedures *push_forward*, *push_down* and *split_and_push*, respectively.

(2) Procedure *push_forward*: If $a[\text{first}(p)] > b$ then all the keys in the packet p must be in $I(\text{forward}(x, l), l)$ provided that all of them were in $I(x, l)$, by Lemma 2.1. By hypothesis, this is indeed the case and the packet must be forwarded replacing the current node of p, x, by $\text{forward}(x, l)$.

(3) Procedure *push_down*: Assume now that $a[\text{last}(p)] \leqslant b$. Then for any key a in p it holds that $a \in I(x, l-1)$, 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.

(4) Procedure *split_and_push*: Finally, b could *hit* the packet, i.e. $a[\text{first}(p)] \leqslant b < a[\text{last}(p)]$. In that case *route* calls procedure *split_and_push*. It first calls *split_packet*, which halves $p = \langle [i..j], x, l, \textit{true}\rangle$ into two packets $p_1 = \langle [i..m], x, l, \textit{true}\rangle$ and $p_2 = \langle [m+1..j], x, l, \textit{true}\rangle$, with $m = (i + j)$ **div** 2. The key b must not hit at least one of p_1 and p_2. Therefore, either it is true that all the keys in p_1 are in $I(x, l-1)$ or all the keys in p_2 are in $I(\text{forward}(x, l), l)$. Hence, one of these 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.

```
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)
    var b: key_type;
    b := key (forward(node(p), level(p)));
    if
        b < a[first(p)] → push_forward(p)
      □ b ⩾ a[last(p)] → push_down(p)
      □ a[first(p)] ⩽ b < a[last(p)] → split_and_push (p, P)
    fi
  end route

  procedure split_and_push (in p: packet; in/out P: set_of_packets)
    var b: key_type; p1, p2: packet; m: 1..k;
    b := key (forward(node(p), level(p)));
    m := (first(p) + last(p)) div 2;
    split_packet (p, m, p1, p2); P := P − {p} ∪ {p1, p2};
    if
        b < a[first(p2)] → push_forward (p2)
      □ b ⩾ a[last(p1)] → push_down(p1)
    fi
  end split_and_push

  var p: packet; P: set_of_packets;
  p := ⟨[1..k], header(S), level(S), true⟩;
  P := {p};

  {begin main loop}
  do active(P) ≠ ∅ →
    for all p: p ∈ active(P) do in parallel
      route(p, P)
    end
  od;
  {end main loop}

  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.

While active(P) $\neq \emptyset$, the *main loop* of the search executes the sentence:

forall p: $p \in$ active(P) **do in parallel** *route*(p, P) **end.**

We call an execution of this **for all** a *stage*. We assume that stages are successively numbered. The first execution of the **for all** is stage 1, the second is stage 2, ... Hence, the notion of a packet p located at (x, l), i.e. the value of its current node/level is (x, l), at the beginning (at the end) of stage t is well defined. Similarly, if a packet p belongs to active(P) before the execution of stage t and route(p, P) pushes it forward, we say that p goes from (x, l) to (forward$(x, l), l)$ during stage t. We should also say that p is forwarded (or moves forward) from (x, l) to its successor during stage t. If p is at (x, l) at the beginning of stage t and *route*(p, P) pushes it down, we say that p at stage t goes (moves down) from (x, l) to $(x, l-1)$ during stage t. Finally, if p is at (x, l) at the beginning of stage t and *route*(p, P) applies *split_and_push* to it, we say that p splits at this stage. Notice that both the set of packets and the set of active packets may change during any stage t. In fact, if we consider the sum of the distances of each of the keys $a[i]$ to their final destinations (measured in number of *push_forward's* and *push_down's*) its value decreases after each stage and when active(P) $= \emptyset$ its value is zero.

3.1. Analysis of read conflicts

During any given stage all active packets perform the statement $b :=$ key(forward(x, l)). Hence, read conflicts arise whenever several packets are at the same node/level (x, l). It is clear that each stage takes time O(1) in a CREW model, but we will like to execute each stage in a EREW PRAM using constant time. We prove in this subsection that there are at most three packets at a given node/level of the skip list in any stage of the search. Later, we show an explicit protocol that allows the execution of each stage in constant parallel time without read conflicts.

First of all, we introduce several useful definitions and notation to study the relationship between packets and their flow through the skip-list. For each packet p, we define two new fields remains(p) and split_side(p). If p has been generated by a split operation during state t at (x, l) and it remains at this same node/level at the end of stage t then remains(p) is true; otherwise, it is false. The value of the split_side field depends on the way the packet has been generated. If p has been generated splitting a packet q and first(p) = first(q) then split_side(p) = right; if last(p) = last(q) then split_side(p) = left. By convention, the split_side of the initial packet $\langle [1..k]$, header(S), level(S), ... $\rangle$ is left. Recall that, except for this first packet, all packets are generated by the split of some other.

Given a packet p and one of its fields f, the value of f of p at the end of stage t will be denoted $f_t(p)$. We assume that stage 0 ends before we enter the main loop of the search for the first time. Let us consider in detail the set of packets located at (x, l) at the end

of stage t:

$$\{p \mid (\text{node}_t(p), \text{level}_t(p)) = (x, l)\}.$$

A packet p can belong to the set above because of two different reasons: either it was generated by the split of a packet located at (x, l) during stage t and p remains in (x, l), or p arrives to (x, l) from another node/level. In the first case $\text{remains}_t(p)$ is true, whilst in the second case $\text{remains}_t(p)$ is false. Hence,

$$\{p \mid (\text{node}_t(p), \text{level}_t(p)) = (x, l)\} = \{p \mid (\text{node}_t(p), \text{level}_t(p)) = (x, l) \wedge \text{remains}_t(p)\}$$
$$\cup \{p \mid (\text{node}_t(p), \text{level}_t(p)) = (x, l) \wedge \neg \text{remains}_t(p)\}.$$

A little more of structure can be added according to the split side of the packets. We introduce the following notations:

$$L_t(x, l) = \{p \mid (\text{node}_t(p), \text{level}_t(p)) = (x, l) \wedge \neg \text{remains}_t(p) \wedge \text{split\_side}_t(p) = \text{left}\},$$

$$M_t(x, l) = \{p \mid (\text{node}_t(p), \text{level}_t(p)) = (x, l) \wedge \text{remains}_t(p)\},$$

$$R_t(x, l) = \{p \mid (\text{node}_t(p), \text{level}_t(p)) = (x, l) \wedge \neg \text{remains}_t(p) \wedge \text{split\_side}_t(p) = \textit{right}\}$$

and have the following partition:

$$\{p \mid (\text{node}_t(p), \text{level}_t(p)) = (x, l)\} = L_t(x, l) \cup M_t(x, l) \cup R_t(x, l).$$

Whenever it is clear from the context, we will omit the explicit mention to the node/level (x, l), thus writing L_t, M_t, etc.

The *subskiplist at* (x, l) of a skip list S, denoted $S(x, l)$, is the skip list of height l, where x acts as a header and $\text{wall}(x, l)$ acts as **NIL**. Node/levels in $S(x, l)$ are those reachable from (x, l). We will denote by $P_t(x, l)$ the set of packets such that its current node/level at the end of stage t belongs to $S(x, l)$. We can keep a record of the packets that have traversed (x, l) from the initial stage up to stage t (see Fig. 2).

- $F_t(x, l)$ contains the set of packets located in the subskiplist $S(\text{forward}(x, l), l)$ at the end of stage t. A packet p belongs to $F_t(x, l)$ iff it has been generated through successive splits from a packet q that was *forwarded* from (x, l) to $(\text{forward}(x, l), l)$ at some stage $t' \leqslant t$. The set $F_t(x, l)$ is defined only if $x \neq \textbf{NIL}$.
- $D_t(x, l)$ contains the packets located in the subskiplist $S(x, l-1)$ at the end of stage t. A packet p belongs to $D_t(x, l)$ iff it has been generated through successive splits from a packet q that was *pushed down* from (x, l) to $(x, l-1)$ at some stage $t' \leqslant t$. The set $D_t(x, l)$ is defined only if $l > 0$.

Formally,

$$D_t(x, l) = P_t(x, l-1), \quad \text{for } l > 0, \quad F_t(x, l) = P_t(\text{forward}(x, l), l), \quad \text{for } x \neq \textbf{NIL}.$$

Note that $P_t = L_t \cup D_t \cup M_t \cup F_t \cup R_t$; moreover, these sets are mutually disjoint. It is immediate to see that each node/level $(x, l) \neq (\text{header}(S), \text{level}(S))$ has a unique *predecessor*, that is, a node/level that precedes it in any search path that passes

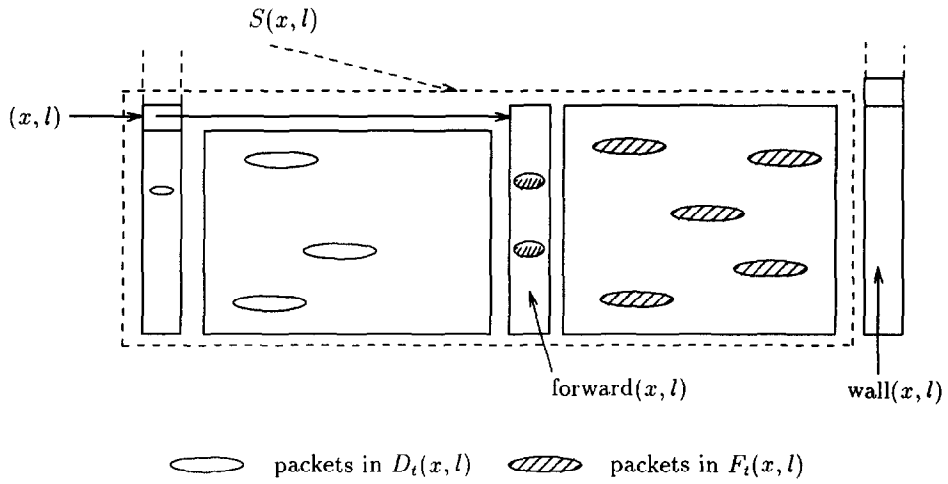

Fig. 2. The sets of packets $F_t(x, l)$ and $D_t(x, l)$.

through (x, l). We denote the predecessor of (x, l) by $\text{pred}(x, l)$. If $l = \text{level}(x)$ then $\text{pred}(x, l) = (y, l)$ where $\text{forward}(y, l) = x$. Then (x, l) can only "receive" packets coming from (y, l) and $F_t(y, l) = P_t(x, l)$. Analogously, if $l < \text{level}(x)$ then (x, l) can only "receive" packets pushed down from $\text{pred}(x, l) = (x, l + 1)$ and $D_t(x, l + 1) = P_t(x, l)$.

In order to compare packets we introduce the following notation:

$$p < q \equiv \text{last}(p) < \text{first}(q),$$

$$p \subseteq q \equiv \text{first}(q) \leqslant \text{first}(p) \wedge \text{last}(p) \leqslant \text{last}(q),$$

$$p \cap q = \emptyset \equiv [\text{first}(p)..\text{last}(p)] \cap [\text{first}(q)..\text{last}(q)] = \emptyset.$$

If $p < q$ we say that p is *at the left* of q (rsp. q is *at the right* of p). A packet p is said to be a *subpacket* of q if $p \subseteq q$. Note that, at the beginning and the end of any stage t, all packets are disjoint, i.e. $p \cap q = \emptyset$, for any pair of different packets p, q in P. Two disjoint packets can be always compared in the sense that either $p < q$ or $q < p$ holds. Moreover, if p and q are disjoint, $q' \subseteq q$ and $p < q$ then $p < q'$. The ordering among packets can be easily extended to sets of packets:

$$A < B \equiv \forall p, q: p \in A \wedge q \in B: p < q.$$

We should note that $<$ is transitive and that both $\emptyset < A$ and $A < \emptyset$ hold for any set of packets A.

Lemma 3.1. *Let p be a packet generated by a split during stage t at (x, l). If p is pushed down or forward during stage t then*

- $\text{split\_side}(p) = \text{left}$ *if and only if p has been pushed forward* ($p \in F_t$).
- $\text{split\_side}(p) = \text{right}$ *if and only if p has been pushed down* ($p \in D_t$).

Proof. Let $b = \text{key}(\text{forward}(x, l))$. Let q be the packet that originates p at stage t. Therefore, $p \subseteq q$ and $a[\text{first}(q)] \leqslant b < a[\text{last}(q)]$. Moreover, as we assume that p is propagated during stage t, either $a[\text{last}(p)] \leqslant b$ or $b < a[\text{first}(p)]$. If $\text{split\_side}(p) = \text{left}$, we have $a[\text{last}(p)] = a[\text{last}(q)]$ and $a[\text{last}(p)] \leqslant b$ cannot hold because this would imply $a[\text{last}(q)] \leqslant b$, thus contradicting our hypothesis that q was split at stage t. Therefore, $b < a[\text{first}(p)]$ should be true and p must have been pushed forward. If $\text{split\_side}(p) = \text{right}$, we have $a[\text{first}(p)] = a[\text{first}(q)]$ and an analogous reasoning proves that p must have been pushed down. If we assume $b < a[\text{first}(p)]$ it follows that $\text{last}(p) = \text{last}(q)$ and hence, $\text{split\_side}(p) = \text{left}$. On the other hand, if $a[\text{last}(p)] \leqslant b$ then $\text{split\_side}(p) = \text{right}$, because necessarily $\text{first}(p) = \text{first}(q)$. □

Lemma 3.2. *For any node/level (x, l) in S and any stage t: (1) if $L_t < D_t \cup M_t \neq \emptyset$, then any packet $p \in L_t$ is pushed down in stage $t + 1$ ($p \in D_{t+1}$): (2) if $\emptyset \neq M_t \cup F_t < R_t$, then any packet $p \in R_t$ is pushed forward in stage $t + 1$ ($p \in F_{t+1}$).*

Proof. Let $b = \text{key}(\text{forward}(x, l))$. Consider the case where $p \in L_t$ and $M_t \neq \emptyset$. As $M_t \neq \emptyset$ there exists a packet q such that $\text{remains}_t(q)$. Moreover, during stage t there was a packet q' at (x, l) such that $q \subseteq q'$ and $a[\text{first}(q')] \leqslant b < a[\text{last}(q')]$. Since $p \in L_t$ the packet p must have been pushed from the predecessor of (x, l) in stage t. Therefore, $p \cap q' = \emptyset$. By hypothesis, $L_t < M_t$ so it follows that $p < q$ and $p < q'$. Since $a[\text{last}(p)] \leqslant a[\text{first}(q')] \leqslant b$, p must be pushed down during the $(t + 1)$th routing step. Assume now that $M_t = \emptyset$, but $D_t \neq \emptyset$. Then, there exists a packet $q \in D_t$ such that $a[\text{last}(q)] \leqslant b$. By hypothesis, $p < q$. Hence, $a[\text{last}(p)] \leqslant a[\text{first}(q)] \leqslant a[\text{last}(q)] \leqslant b$ and p must be pushed down in stage $t + 1$. For the second claim, when $p \in R_t$, the proof goes along the same lines. □

Lemma 3.3. *Let (x, t) be a node/level such that at the end of stage t, each of L_t, M_t, R_t has at most one packet and $L_t < M_t < R_t$. Then, at most two packets can be propagated from (x, l) to one of its adjacent node/levels in stage $t + 1$; moreover, if two packets are pushed from (x, l) to the same node/level then they must have different* split_side.

Proof. We prove the lemma assuming that $M_t \neq \emptyset$. The case where $M_t = \emptyset$ can be proved using a similar reasoning. Let $b = \text{key}(\text{forward}(x, l))$, as usual. By the hypothesis of the lemma and our initial assumption, M_t should be a singleton: $M_t = \{p_m\}$. As $\text{remains}_t(p_m)$ is true, the packet p_m was generated at (x, l) during stage t from the split of some other packet p'_m. Therefore, $p_m \subseteq p'_m$ and $a[\text{first}(p'_m)] \leqslant b < a[\text{last}(p'_m)]$. Assume that $L_t = \{p_l\}$. By Lemma 3.2, the packet p_l will be pushed down to $(x, l - 1)$ in stage $t + 1$ and $\text{split\_side}(p_l) = \text{left}$ does not change. Similarly, if $R_t = \{p_r\}$ then p_r will be pushed to $(\text{forward}(x, l), l)$ in stage $t + 1$ and $\text{split\_side}(p_r) = \text{right}$. Let us consider now the evolution of p_m in stage $t + 1$. There are two possibilities: either p_m is split or it is propagated. If p_m splits into p_{mr} and p_{ml} with $\text{split\_side}(p_{mr}) = \text{right}$ and $\text{split\_side}(p_{ml}) = \text{left}$, one of the following relations holds: $a[\text{last}(p_{mr})] \leqslant b$ or

$b < a[\text{first}(p_{ml})]$. Then one of these subpackets will be pushed forward or down. If p_{mr} is pushed down to $(x, l-1)$ we can conclude that at most one packet, p_r, was pushed forward; and at most two packets, p_l and p_{mr}, have been pushed down during stage $t+1$, their split_sides being different: $\text{split\_side}(p_l) = \text{left} \neq \text{split\_side}(p_{mr}) = \text{right}$. If, one the contrary, p_{mr} remains at (x, l) and p_{ml} is pushed forward then we conclude that at most two packets, p_r and p_{ml}, have been forwarded to the same node/level with $\text{split\_side}(p_r) = \text{right} \neq \text{split\_side}(p_{ml}) = \text{left}$. Consider now what happens when p_m is not split in stage $t+1$. Then p_m should be at the left of b (case $a[\text{last}(p_m)] \leqslant b$) with $\text{split\_side}(p_m) = \text{right}$ or at the right of b case $b < a[\text{first}(p_m)]$ with $\text{split\_side}(p_m) = \text{left}$. In both cases p_m has a different split_side than the other packet (p_l or p_r) that could be pushed to the same node/level. □

Lemma 3.4. *For any node/level (x, l) in S and any stage t, each one of the sets L_t, M_t and R_t has at most one packet. Moreover, it also holds $L_t < D_t < M_t < F_t < R_t$.*

Proof. We prove the lemma by induction on the number of stages t.

Inductive basis ($t = 0$): At the end of stage $t = 0$, we only have a packet p pointing to (header(S), level(S)). Furthermore, $\text{remains}_0(p) = \text{false}$ and $\text{split\_side}_0(p) = \text{left}$. The lemma trivially holds since $L_0(\text{header}(S), \text{level}(S)) = \{p\}$ and all other sets are empty.

Inductive step ($t \Rightarrow t+1$): Assume $M_t \neq \emptyset$. Then, by induction hypothesis, M_t has exactly only one packet p_m. Lemma 3.3 guarantees that at most two packets, say pp_l and pp_r, come from $\text{pred}(x, l)$ during stage $t+1$. As they have different split_side we can assume $\text{split\_side}(pp_l) = \text{left}$ and $\text{split\_side}(pp_r) = \text{right}$. When $L_t = \{p_l\}$, the induction hypothesis and Lemma 3.2 imply $p_l \in D_{t+1}$. In that same stage $t+1$ the packet pp_l comes from $\text{pred}(x, l)$. Since $\text{split\_side}(pp_l) = \text{left}$ and p_l has been pushed down, we have $L_{t+1} = \{pp_l\}$. Note that $L_{t+1} = \emptyset$ if no packet is pushed from $\text{pred}(x, l)$ with split_side = left, no matter there existed a packet p_l in L_t or not. By a similar argument, if $R_t = \{p_r\}$ then $p_r \in F_{t+1}$ and $R_{t+1} = \{pp_r\}$. In the case that such pp_r does not exist, then $R_{t+1} = \emptyset$. The packet p_m that belongs to M_t will be pushed or split in stage $t+1$ into two packets and only one of them would remain at (x, l). Therefore, $M_{t+1} = \emptyset$ if p_m moves to some other node/level and $M_{t+1} = \{p'_m\} \neq \emptyset$ when p_m splits, where $p'_m \subseteq p_m$ is the subpacket that remains at (x, l). Therefore, when $M_t \neq \emptyset$ the node/level(x, l) can contain at most the packets pp_l, p'_m and pp_r at the end of stage $t+1$ and consequently of L_{t+1}, M_{t+1} and R_{t+1} has at most one packet.

We now analyze the relation between the packets in P_t and the packets pp_l and pp_r that move from $pred(x, l)$ to (x, l) in stage $t+1$. For each one of pp_l and pp_r, there are two possibilities: either it is generated during stage $t+1$ at $\text{pred}(x, l)$ or it is not. If pp_l is not generated by a split in stage $t+1$ and since we assume that $\text{split\_side}(pp_l) = \text{left}$ it follows that $pp_l \in L_t(pred(x, l))$. The induction hypothesis, now applied to $\text{pred}(x, l)$, yields $\{pp_l\} < D_t(\text{pred}(x, l)) \cup F_t(pred(x, l))$. If $\text{pred}(x, l) = (x, l+1)$ then pp_l has to be pushed down and $P_t(x, l) = D_t(pred(x, l))$; otherwise,

pp_l has to be pushed forward and $P_t(x, l) = F_t(pred(x, l))$. In both cases, it holds $\{pp_l\} < P_t(x, l)$. If pp_l is generated by the splitting of a packet q during stage $t + 1$ it cannot be pushed down, i.e. $pred(x, l) \neq (x, l + 1)$, because it would contradict Lemma 3.1 otherwise. Hence, forward $(pred(x, l), l) = (x, l)$. Since $\emptyset \neq M_t(x, l) \subseteq F_t(pred(x, l)) < R_t(pred(x, l))$, Lemma 3.2 implies $q \in L_t(pred(x, l)) \cup M_t(pred(x, l))$. Applying once again the induction hypothesis, we have

$$pp_l \subseteq q \in L_t(pred(x, l)) \cup M_t(pred(x, l)) < F_t(pred(x, l)) = P_t(x, l),$$

so $\{pp_l\} < P_t(x, l)$ is also true if pp_l is generated by a split in stage $t + 1$. A similar analysis yields $P_t(x, l) < \{pp_r\}$, whether pp_r is generated in stage $t + 1$ or not.

Let us consider the relationship between $L_{t+1}(x, l)$ and $D_{t+1}(x, l)$. This last set contains: the packets (or subpackets of these) already in D_t, the packet in L_t if there were one and, in some cases, the packet or a subpacket of the packet in M_t. Since all packets in D_{t+1} were also in P_t or are subpackets of packets in P_t and $\{pp_l\} < P_t$, it follows that $L_{t+1} < D_{t+1}$. Moreover, the inductive hypothesis implies $D_t < M_t$, and it is easy to show that $D_{t+1} < M_{t+1}$, irrespective of the fact that p_m moves or splits. Of course, the relationships above trivially hold unless the sets are nonempty. It is not difficult to show, using similar arguments, that $M_{t+1} < F_{t+1} < R_{t+1}$ since $P_t < \{pp_r\}$. Hence, if $M_t(x, t) \neq \emptyset$

$$L_{t+1} < D_{t+1} < M_{t+1} < F_{t+1} < R_{t+1}.$$

The lemma also holds when $M_t(x, l) = \emptyset$. The proof in this case is rather similar to the one already shown and independently considers the cases $D_t \cup F_t \neq \emptyset$ and $D_t \cup F_t = \emptyset$. □

From Lemma 3.4 it is plain to see that if we use the procedure *exclusive_read* given in Algorithm 2 to access the forward pointers and the keys of the nodes they point to, each routing step can be done in constant time avoiding read conflicts.

Lemma 3.5. *The procedure route(p, P) can be implemented in the EREW model in constant parallel time using k processors.*

---

procedure exclusive_read (**out** b: key_type; **in** p: packet)
  if remains(p) → b := key(forward(node(p), level(p))) **fi**;
  if ¬remains(p) ∧ split_side (p) = left → b := key (forward(node(p), level(p))) **fi**;
  if ¬remains(p) ∧ split_side(p) = right → b := key (forward(node(p), level(p))) **fi**
end exclucive_read

Algorithm 2. Exclusive read of keys.

3.2. *Efficiency of the search algorithms*

We analyze in this subsection the expected number of routing steps or stages in a search. The performance of the parallel search is stated in Theorem 3.1, at the end of the section. It follows directly from the results of the previous subsection and results in this one.

Before we consider the expected number of stages, we should note that we check for active packets before the execution of each stage or routing step: (**do** $active(P) \neq \emptyset$). If there is at least one active packet, each processor locally "decides", in constant time, whether its associated packet has to be routed or not (see Algorithm 3a):

for all $p: p \in active(P)$ **do in parallel**...

Note that the status of a packet (active/inactive) is not the status of the processor associated with that packet. When we check if there is any active packet, the processors associated with the packets perform a parallel **or** with the active status of all the packets. Since this check is made before the start of each new stage, each stage would have cost $O(\log k)$ and the whole search would have expected cost

$O(\log k \cdot$ expected number of stages$)$.

```
...
do active(P) ≠ ∅ →
  for all p: p ∈ active(P) do in parallel
    route(p, P)
  end
od
...
```

Algorithm 3a. Main loop (Algorithm 1).

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

Algorithm 3b. Main loop of the search by runs.

To avoid this $O(\log k)$ cost per stage we execute *runs* of $\lceil L(n) \rceil + \lceil \log_2 k \rceil$ stages, where $L(n) = \log_{1/p} n$. 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 $\lceil L(n) \rceil + \lceil \log_2 k \rceil$ stages is executed, etc. (see Algorithm 3b). We prove in Lemma 3.6 that the expected number of stages is $O(\log n + \log k)$; hence, the expected number of runs is constant and the expected performance is $O(\log n + \log k)$. Note that we assume that stages can be executed even if there are not active packets at all: the host computer issues each of the instructions in the *route* procedure to each processor, but they execute **skips** or NOPs instead.

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, $\Pr\{X > t\} \leqslant \Pr\{Y > t\}$. We write $X \leqslant_{\text{prob}} Y$ [25]. Note that $X \leqslant_{\text{prob}} Y$ implies $E(X) \leqslant E(Y)$. We use $B(n, p)$ 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* $NB(r, p)$. It is equal to the number of failures seen before the rth success in a series of random independent trials, where the probability of success in a trial is p. If $r = 1$, the random variable is said to be geometric. Finally, to avoid the use of ceiling function we assume that $\log_2 k$ is an integer number.

Lemma 3.6. *Let $C_{n,k}$ be the random variable denoting the number of stages needed to search for k keys in 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 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, $C_{n,k} = \max_{1 \leqslant i \leqslant k}\{C_i\}$. The number of *push_down* operations applied to the packets containing a given key $a[i]$ is bounded by level(S), the height of the skip list. The expected height $E(H_n)$ for random skip lists of size n is $\Theta(\log n)$ [21]. Similarly, the number of *push_forward* operations applied to packets containing a given key is bounded by the *width* of the skip list. The width $W(S)$ of a skip list S is the maximum number of forward pointers followed on any search path. Let W_n denote the random variable corresponding to the width of a random skip list of size n.

Devroye [6] has shown 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,

$$C_{n,k} \leqslant_{\text{prob}} H_n + W_n + \log_2 k,$$

$$E(C_{n,k}) \leqslant E(H_n) + E(W_n) + \log_2 k = O(\log n + \log k). \qquad \square$$

Our next lemma states that significant derivations between the expected number of stages and the actual number of stages are unlikely to occur. The lemma is proved using Chernoff bounds. Similar statements have been proved for the time complexity of the sequential search algorithm in skip lists and for the storage requirements of skip lists [27]. Let us recall that, using Chernoff tail bound lemma [5], for a binomial random variable $X = B(n, p)$, any value a and any $t > 0$

$$\Pr\{X \geqslant a\} \leqslant (q + pe^t)^n e^{-ta}, \quad \Pr\{X \leqslant a\} \leqslant (q + pe^{-t})^n e^{ta}, \quad q \equiv 1 - p. \tag{1}$$

For a negative binomial random variable $Y = NB(r, p)$, we have the following bound for any $a \geqslant 1$ and $t > 0$

$$\Pr\{Y \geqslant a\} \leqslant e^{-ta}\left(\frac{pe^t}{1 - qe^t}\right)^r, \qquad q \equiv 1 - p. \tag{2}$$

Lemma 3.7. *Let $C_{n,k}$ be the number of stages in a search for k keys in a random skip list of size n. Then, for any $\theta > c$ and sufficiently large n, there exists $\gamma = \gamma(\theta) > 0$ such that*

$$\Pr\{C_{n,k} \geqslant \theta L(n) + \log_2 k\} = \Theta(n^{-\gamma}),$$

where $c = c(p)$ is the unique solution, larger than $1/p$, of the equation

$$\log(1 - p) = \log(c - 1) - \frac{c}{c - 1}\log c.$$

Proof. This proof essentially follows the argument given by Devroye in [6]. Let D_k denote the length of the search path from the topmost level of the header to the kth item in a random skip list. Then $\max_{1 \leqslant k \leqslant n} D_k = H_n + W_n$. It turns out that each of the D_k is stochastically smaller than D_n and hence, for any z

$$\Pr\{C_{n,k} \geqslant z + \log_2 k\} \leqslant n \Pr\{D_n \geqslant z\}. \tag{3}$$

Let $X_1 = NB(L(n) - 1, p)$, $X_2 = NB(1, 1 - p)$ and $X_3 = B(n, 1/np)$. The random variable $L(n) + X_1 + X_2 + X_3$ is an stochastic upper bound for D_n, the path length to the last item in a random skip list of size n [25].

We now define $z := \theta L(n)$, with $\theta > 1$. Applying the Chernoff bounds above (Eqs. (1) and (2)) we have

$$\Pr\{X_2 \geqslant z\} \leqslant \left(\frac{1 - p}{p^{\varepsilon/\theta} - p}\right) n^{-\varepsilon} = \Theta(n^{-\varepsilon}),$$

$$\Pr\{X_3 \geqslant z\} \leqslant \exp(p^{-1}(p^{-\varepsilon/\theta} - 1))\, n^{-\varepsilon} = \Theta(n^{-\varepsilon}),$$

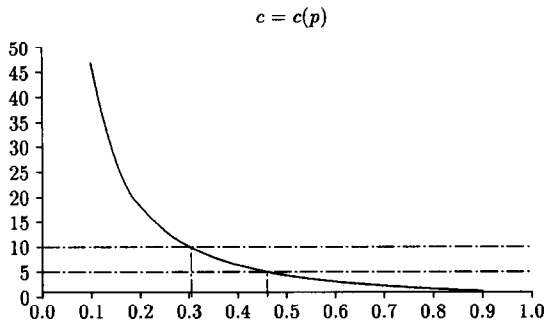

Fig. 3. The value of c as a function of p.

where for both inequalities $\varepsilon > 0$ is any arbitrary constant and we take $t = (\varepsilon/\theta)\ln(1/p) > 0$ (however, one should avoid $\varepsilon = \theta$ for obvious reasons in the first inequality).

Finally, let $\ell := L(n) - 1$ and $u := \ell/z$. For X_1 we have

$$\Pr\{X_1 \geqslant z\} \leqslant e^{-tz}\left(\frac{pe^t}{1-qe^t}\right)^{\ell} \leqslant p^{\ell}q^{z-\ell}\frac{z^z}{\ell^{\ell}(z-\ell)^{z-\ell}}$$

$$= \left\{\left(\frac{q}{1-u}\right)^{1-u}\left(\frac{p}{u}\right)^{u}\right\}^{z}.$$

where we take $e^t = (z-\ell)/qz$ (and hence θ must be larger than $1/p$ to ensure that $t > 0$).

The lemma follows from Eq. 3 and the bounds for the tails of X_1, X_2 and X_3:

$$n\Pr\{D_n \geqslant \theta L(n)\} \leqslant n\Pr\{X_1 + X_2 + X_3 + L(n) \geqslant \theta L(n)\}$$

$$\leqslant \Theta(n^{1-\varepsilon}) + \exp\left(L(n)\left(\ln(1/p) + \theta\left((1-u)\ln\frac{q}{1-u} + u\ln\frac{p}{u}\right)\right)\right)$$

$$= \Theta(n^{-\gamma}),$$

where $-\gamma$ is the coefficient of $L(n)$ and we make ε arbitrarily large. To complete the proof, recall that for n large enough $u \to 1/\theta$ and therefore, the constant γ is positive if θ is larger than the unique root $c > 1/p$ of the equation:

$$(c-1)\ln(1-p) - (c-1)\ln\frac{c-1}{c} + \ln c = 0. \qquad \square$$

From Lemmas 3.5–3.7, and the fact that the current node of each packet at the end of the main loop can be optimally spread (segmented copy) in time $O(\log k)$ using $k/\log k$ processors [3, 26], we have the next Theorem:

Theorem 3.1. *The parallel search in a skip list can be implemented in a EREW PRAM model with expected time* $O(\log n + \log k)$ *and using* k *processors, where* k *is the*

number of keys to be searched for and n is the size of the skip list. Moreover, the probability that its performance deviates from its expected value decreases as $O(n^{-\gamma})$, *for some* $\gamma > 0$ *that depends on P and the amount of deviation.*

4. Insertion

Assume, for the sake of simplicity and without loss 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 4). First, the procedure *search_with_update* searches for the k keys and builds a collection of *update* arrays with information about the would-be predecessors of the new items. Second, *create_new_nodes* allocates k new nodes to hold the items to be inserted. Third, *make_succ* builds an *skip array* called *succ*. This array, together with the *update* arrays, provides information about the predecessor and successor of each node/level to be inserted. Fourth, the procedure *merge* actually inserts the k new nodes in the appropriate places of the skip list.

(1) Procedure *search_with_update*: Upon termination a set of *update* arrays, one for each new item, has been computed. The *update* array for the ith new item has information about its would-be predecessors for each level. We have

$$\begin{aligned} update[l, i] = \text{“the unique node } x \text{ in } S \text{ such that} \\ \text{key}(x) < a[i] \leqslant \text{key (forward}(x, l))\text{”}, \\ 1 \leqslant l \leqslant \text{level}(S), \quad 1 \leqslant i \leqslant k. \end{aligned}$$

Note that $update[l, i]$ will be predecessor of the ith new item at level l only if a level larger or equal to l is assigned to the new item and there is not any key to be inserted, say $a[j]$, such that $\text{key}(update[l, i]) < a[j] < a[i]$. To compute these *update* arrays we maintain an *update* array for each packet p. Let p be a packet that has been generated at node/level (x, l). We keep track of the path followed by p in its update(p) array as follows:

$$\begin{aligned} \text{update}(p)[k] = \text{rightmost node of level } \geqslant k \text{ in the search path from} \\ (x, l) \text{ to the current node/level } (x', l') \text{ of } p, \quad l' < k \leqslant l. \end{aligned}$$

In fact, the *update* array of a packet p is the one associated to its first key. To collect the information in the *update* arrays, the procedure *push_down* is modified by adding

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

just before we decrement the current level of p.

When the search loop finishes we must spread the *update* arrays of the packets to all the keys. If the key $a[i]$ was in a packet p the last time that key was at level l, then the lth component of the update array of p should be copied to $update[l, i]$. This is not very difficult since the needed value is held, by construction, by $update[l, j]$, where j is the maximum index such that $j \leqslant i$ and $update[l, j] \neq$ **NIL**.

```
procedure insert (in a: array [1..k] of key_type; in/out S; skip_list)
  type skip_array = array [1..??, 0..k + 1] of 0..k + 1;
                                        {number of rows is computed at run-time}
    update_array = array [1..??, 1..k] of refnode_type;
    node_array = array [1..k] of refnode_type;
    level_array = array [0..k + 1] of integer;
  procedure search_with_update (...);
  procedure create_new_nodes (...);
  procedure make_succ (in level: level_array; in m: integer; out succ: skip_array)
    var i: 0..k, l: integer;
    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[l, i] := min(j: i < j ⩽ k + 1 ∧ level[j] ⩾ l)
      end
    end
  end make_succ

  procedure merge(...); {see Algorithm 5}
  var succ: skip_array;
    update: update_array;
    node: node_array;
    level: level_array;
    m: integer;
  search_with_update (a, S, 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 4. Insertion algorithm.

(2) Procedure *create_new_nodes*: To create the new nodes for the new k items, k random levels are independently generated in the first step. This is achieved by making each processor call, in parallel, a function *random_level*. The value it returns follows a geometric $NB(1, p)$ distribution. The random level of the ith new item is stored in $level[i]$. Finally, the procedure allocates k new nodes with as many forward pointers per node as its level indicates and stores each of the k keys in its corresponding node. It returns an array *node* of pointers to these new nodes, as well as the array *level*.

(3) Procedure *make_succ*: It builds the skip array *succ*. This array mimics a new skip list corresponding to the keys in a alone. A skip array is a two-dimensional array with $m = \max\{level[i]: 1 \leqslant i \leqslant k\}$ rows and $k + 2$ columns. Any column i with $1 \leqslant i \leqslant k$ has $level[i]$ useful rows. Columns 0 and $k + 1$ are sentinels with m rows. The array *succ*, for $0 \leqslant i \leqslant k$, is defined as follows:

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

Recall that we must be able to decide whether $update[l, i]$ is the actual predecessor of the node $node[i]$ at level l or not. The skip array *succ* brings exactly this information (see the description of the procedure *merge*). Note that $succ[l, 0]$ is the index of the first new node whose level is at least l. All the rows of *succ* can be computed independently and can be filled using easy variants of the tree or parallel prefix computations [3, 26].

(4) Procedure *merge*: This phase merges *succ* with S (see Algorithm 5). If $m >$ level(S), we fill with pointers to header(S) the components level(S) up to m 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 a node $node[i]$ with corresponding key $a[i]$ and $level[i] \geqslant l$. 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 5). We consider only two of them, the others are similar. Given i with $1 \leqslant i \leqslant k$, call $succ[l, i] = j$ and suppose that

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

As j is different from the sentinel $k + 1$ there exists a node $node[j]$ such that its level is at least l and $a[j] > a[i]$. Moreover, as $update[l, i] \neq update[l, j]$ there exist nodes in S at level l whose keys lie between $a[i]$ and $a[j]$. Hence, $node[j]$ is not the successor of $node[i]$ at level l in the new skip list and $update[l, j]$ is the actual predecessor of $node[j]$. The successor of $node[i]$ at level l is the node that was the successor of update$[l, i]$ at level l. The following assignments are thus needed:

forward(node$[i]$, l) := forward ($update[l, i]$, l);

forward($update[l, j]$, l) := $node[j]$;

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

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

Algorithm 5. Procedures merge and merge_level.

Now assume that $j \neq k+1$, but $update[l, i]$ and $update[l, j]$ coincide. Then we can be sure that $update[l, j]$ will not be the actual predecessor of $node[j]$; $node[i]$ will be the predecessor of node$[j]$ at level l. The procedure *merge_level* performs in this case the assignment:

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

All the tests and pointer updates can be done in parallel constant time, since only the local information provided by the *update* and *succ* arrays is needed.

4.1. Analysis 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 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. The initialization of the *update* arrays components to **NIL** can be done by k processors in expected time $O(\text{level}(S)) = O(\log n)$. The information collected in these *update* arrays must be spread to all the processors when the main loop of the search finishes; 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 in an EREW PRAM. We can allocate $k/\text{level}(S)$ processors for each level in the *update* arrays, and do the broadcasting of the *update* arrays in time $O(\text{level}(S) + \log(k/\text{level}(S)))$ using k processors. Since the expected level of S is $O(\log n)$ the expected time to spread the update arrays is $O(\log n + \log k)$.

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 proportional 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 new item has been independently computed. Later, 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 [15, 22, 23].

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 processors. Using k/m processors for each row, it takes $O(m + \log(k/m))$ parallel time to fill all the rows of *succ*, using a slight variation of common parallel prefix computations. Since the expected value of m is $O(\log k)$, this part of the insertion has expected cost $O(\log k)$ with k processors. In order to avoid concurrent reads in the *level* array, the *make_succ* procedure should fill a bidimensional array of bits, say *lb*, such that $lb[l, i] = 1$ if and only if $level[i] \geqslant l$. The array *lb* can be filled by k processors in expected time $O(\log k)$ (the expected number of rows in *lb*) in an EREW model. Then, the computation of each row of the *succ* array can be done using the corresponding row of the *lb* array and concurrent read conflicts are avoided.

Finally, a call to the procedure *merge* is performed. The execution of this procedure with k processors takes parallel expected time $O(\log k)$, since *merge_level* has cost $O(1)$ using k processors and *merge* calls *merge_level* m times, once for each level. It is not difficult to see that the procedure *merge_level* can be written without using concurrent reads. Taking all these contributions into account Theorem 4.1 follows. Moreover, we can prove that there is a very small probability of bad performance for the insertion algorithm using Chernoff bounds, following arguments similar to those used in Section 3.2. In particular, an $O(n^{-\gamma})$ bound follows from the search phase. The phase where new nodes are created needs $O(\log k)$ in the average. Both the construction of the skip array and the merge phases take $O(\log k)$ expected time. In

both cases the performance is proportional to the height of a random skip list of k elements. Then, it is not difficult to prove, using Chernoff bounds as in Section 3.2, that the probability of large deviations from the expected time is $O(k^{-\gamma})$. Furthermore, the cost of the search phase is independent of that of the next phases.

Theorem 4.1. *The insertion algorithm for skip lists can be implemented in a EREW PRAM model with expected time* $O(\log n + \log k)$ *using* 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 its expected value decreases as* $O(n^{-\gamma} + k^{-\gamma})$, for some $\gamma > 0$.

5. Deletion

We now consider the deletion algorithm. First, it uses the *search_with_update* procedure to find where are the keys to be deleted and its predecessors at each level. After this step, it constructs three skip arrays: *succ*, *pred* and *last*, giving information about successors, predecessors and blocks of consecutive nodes to be deleted. For each node to be removed and each of its levels, we must know whether its predecessor and successor will remain or be also removed. Which one of these cases holds can be checked in parallel using the *update* and the three mentioned skip arrays. Each case can be managed with simple parallel address arithmetic techniques similar to those used in the insertion algorithm.

Both the procedures *search_with_update* and *make_succ* are the ones that we already described for the insertion. The number of rows in the *succ* array is computed from the actual levels of the items to be deleted. We now give a short description of the other procedures used in the deletion algorithm (Algorithm 6).

(1) Procedure *make_pred*: This procedure computes a skip array called *pred*, that provides information on the predecessors of the items to be deleted, given their levels.

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

The procedure is quite similar to the procedure *make_succ*.

(2) Procedure *make_last*: Consider a node $node[i]$ to be deleted and a level $l \leqslant \text{level}[i]$. Maybe a block of consecutive node/levels in the list of level l and starting from $(node[i], l)$ will be deleted too. In order to chain the nodes that will not be deleted, we need to know the index of the rightmost element in this block. We call it $last[l, i]$. Therefore, the nodes

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

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

$$last[l, i] = \text{minimum index } j,\ i \leqslant j \leqslant k, \text{ such that } level[j] \geqslant l \text{ and } node[j] \text{ will be deleted but forward}(node[j], l) \text{ will not.}$$

```
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: integer; in node: node_array;
                       in succ: skip_array; out last: skip_array)
    var i: 0..k; l: integer;
    for all l: 1 ⩽ l ⩽ m do in parallel
      for all i: 1 ⩽ i ⩽ k ∧ level[i] ⩾ l do in parallel
        if
            succ[l, i] = k + 1 → last[l, i] := i
          □ succ[l, i] ≠ k + 1 → last[l, i] := min(j : i ⩽ j ⩽ k ∧ level[j] ⩾ l ∧
                                                  forward (node[j], l) ≠ node [succ[l, j]])
        fi
      end
    end
  end make_last

  procedure remove (...) {see Algorithm 7}
  var pred, succ, last : skip_array;
    update : update_array;
    node: node_array;
    level : level_array;
    m : integer;
    i : 1..k;
  search_with_update (a, S, 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, succ, last);
  remove (pred, last, update, node, m, S)
end delete
```

Algorithm 6. Deletion algorithm.

There are two cases in the computation of $last[l, i]$. When $succ[l, i] = k + 1$, it means that node$[i]$ is the last whose level is greater or equal to l and will be deleted. In particular, the forward pointer of node$[i]$ at level l points to a node that will not be

removed from S. Therefore, $last[i, l] = i$. When $succ[l, i] \neq k + 1$ we have

"forward$(node[j], l)$ will not be deleted from S"

$\equiv$ "forward$(node[j], l) \neq node[succ[l, j]]$)"

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

$$last[l, i] = min(j: i \leqslant j \leqslant k \wedge level[j] \geqslant l \wedge forward(node[j], l) \neq node[succ[l, j]])$$

(3) Procedure *remove*: It actually removes the elements from the skip list. Given a $node[i]$ to be deleted and level l, its predecessor at level l is $update[l, i]$. There are two possible cases. The first one happens when $update[l, i]$ has also to be removed. This can be easily checked because

"$update[l, i]$ will be deleted" $\equiv$ "$pred[l, i] \neq 0 \wedge update[l, i] = node[pred[l, i]]$".

As $node[pred[l, i]]$ will also be deleted the pointer forward $(node[pred[l, i]], l)$ is redundant and the algorithm does not take care of it. The second one occurs if $update[l, i]$ has not to be removed from S:

"$update[l, i]$ remains" $\equiv$ "$pred[l, i] = 0 \vee update[t, i] \neq node[pred[l, i]]$".

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

The analysis of the deletion algorithm goes along the same lines as that of the insertion algorithm. The searching phase needs $O(\log n + \log k)$ expected time with k processors. The computation of the skip arrays *succ*, *pred* and *last* reduces to parallel tree computations with expected cost $O(\log k)$ using k processors. Finally, each call to *remove_level* consumes constant parallel time using k processors; the procedure is called $O(\log k)$ times on the average (see Algorithm 7). Summing up these contributions the next theorem follows.

Theorem 5.1. *The deletion algorithm for skip lists can be implemented in an EREW PRAM model with expected time* $O(\log n + \log k)$ *and using k processors, where k is the number of keys to be deleted and n is the length of the skip list. Moreover, the probability that the performance of the deletion algorithm significantly deviates from its expected value decreases as* $O(n^{-\gamma} + k^{-\gamma})$, for some $\gamma > 0$.

6. Implementation issues

We discuss in this section several implementation-related issues and give some examples using the programming language C*. A preliminary implementation of the search and insertion algorithms has already been developed for the Connection Machine CM-2 and their performance has been empirically studied [19].

```
procedure remove (in pred, last: skip_array; in update: update_array;
                  in node: node_array; in m: integer; in/out S: skip_list)
  procedure remove_level (in pred, last: skip_array; in update: update_array;
                          in node: node_array; in l: integer)
    var i: 1..k; j: 0..k;
    for all i: 1 ≤ i ≤ k ∧ l ≤ level [i] do in parallel
      j := pred[l, i];
      if
          j = 0 ∨ update [l, i] ≠ node [j] →
                  forward(update [l, i], l) := forward (node [last [l, i]], l)
        □ j ≠ 0 ∧ update [l, i] = node [j] → skip
      fi
    end;
    for all i: 1 ≤ i ≤ k do in parallel
      free (node[i])
    end
  end remove_level
  var l: integer;
  for l: 1 ≤ l ≤ m do
    remove_level(pred, last, update, node, l)
  end;
  do forward (header(S), level(S)) = NIL →
            level(S) := level(S) − 1
  od
end remove
```

Algorithm 7. Procedures remove and remove_level.

C* is a data parallel language for the Connection Machine CM-2 system [15, 28]. In the data parallel model of C*, there are many tiny processors, each one capable of storing local data, performing computations and communicating with other processors. Parallel instructions are synchronously issued from the front end to all the processors; each processor can execute the instruction using its own locally stored data or just wait for the next instruction; a status bit controls the behavior of each processor. *Shapes* and *parallel variables* are basic notios of C*. A shape explains how to configure the data in the processors. For instance, a shape declaration like

```
shape [128] my_vector;
```

tells that my_vector consists of 128 positions numbered 0, 1, ..., 127 arranged in a linear fashion. There will be a processor associated to each position and each processor can be uniquely identified by its position i. The shape my_vector is said to

have rank 1; bidimensional shapes (of rank 2) and k-dimensional shapes are also possible. Any C* parallel variable has a type and a shape. The following declaration

```
int a_vector:my_vector;
```

introduces a new parallel variable a_vector with shape my_vector and where each position stores an integer. Just as common arrays can be created at runtime in C, both shapes and parallel variables can be dynamically allocated by means of the library functions allocate_shape and palloc. Shapes and parallel variables can be accessed through pointers, passed as parameters to functions, etc.

In any moment of the execution of a C* program, there must be a *current shape*. Computation and communication can only be done between parallel variables of the current shape. The clause with sets the current shape.

Standard C operators such as =and+= have been given new meanings. For instance, if both p, q and r are parallel variables of shape my_vector, the statement p=q+r; is equivalent to

for all $i:0 \leqslant i < 128$ **do in parallel**
  $p[i] := q[i] + r[i]$
end

The operator += (and others of the type $op=$) are parallel reduction operators in C*. For instance, if s is an scalar int, the statement s+=p; does in parallel the following

```
s=s+[0]p+[1]p+···+[127] p;
```

where [i]p denotes the ith element of the parallel variable p (this notation is called *left-indexing*). Besides operators such as += (parallel sum) and |= (parallel logical or), C* offers library functions such as scan that performs sophisticated parallel prefix computations [3].

General communication between processors is possible using *send* or *get* operations. Consider two parallel variables dst and src of shape my_vector and

```
int address:my_vector;
```

Assume furthermore that all values of the parallel variable address are in the range 0..127 and different, i.e. address stores a permutation. The get operation dst=[address]src; is equivalent to

for all $i:0 \leqslant i < 128$ **do in parallel**
  $dst[i] := src[address[i]]$
end

The send operation [address]dst=src; is equivalent to

for all $i: 0 \leqslant i < 128$ **do in parallel**
  $dst[address[i]] := src[i]$
end

A function worth mentioning in this context is pcoord(j) which evaluates to the index of each position along the jth dimension. For a linear shape such as my_vector, the evaluation of pcoord(0) at its ith position returns i.

Finally, processors can be activated and deactivated at will before executing any parallel computation or communication instruction. The clause **where** serves this purpose. In a **where** clause, each (active) processor evaluates a condition; if the result is false, the processor becomes inactive until the **where** clause ends. For instance,

```
with(my_vector)
where(q>=0)
p=q+r;
```

performs the parallel assignment only for those positions i such that [i]q is positive.

The notions of packets and packet routing are useful to explain the search algorithm, but it is not easy to express these notions in most current parallel languages. A simple way to implement Algorithm 1 is to define a parallel array of size k, so each processor holds a key, an active status, etc. For instance, in C*, we can dynamically declare a shape of size k using allocate_shape and then a parallel variable of that shape with palloc, where each component of the parallel variable is a record with several fields. The field key holds the keys to be searched for. The other fields are: active, first, last, node, level, remains and split_side. We think of each processor containing a *slice* associated with a particular key: the ith processor contains the ith key and the ith value for each of the other fields.

Consider a packet p that holds the keys $a[i], \ldots, a[j]$ (first(p) = i and last(p) = j) at some stage of the search algorithm. We use the convention that the processor associated to the first key of a packet is "responsible" for the whole packet. This situation is represented by storing first(p) and last(p) in the ith components of the fields first and last, and storing the current node/level of the packet in the ith components of the fields node and level. If p is an active packet, then the value of the ith component of the active field is 1 (true), whereas the rest of the components of the active field that are associated to other keys in the same packet p are set to 0 (false). If p is not active, the active field is set to 0 for all keys within p.

We give some useful C* declarations for the search procedure:

```
shape []packet_shape; /* number of positions in the shape is left
                         unspecified; it will be specified at run-time */
  typedef struct {
```

```
  key_type key;
  int first, last, level;
  refnode_type node;
  bool active, remains, split_side; }search_elem;
typedef search_elem: packet_shape search_array;
```

We now sketch the C* code for the search procedure. It takes as arguments a pointer to a search_array and a skip list. We assume that the key field of the search_array is already initialized with the k keys. The results is returned through the node field.

The first lines of code create a single active packet located at (header(S), level(S)). The test for active packets can be accomplished by performing a parallel **or** reduction (|=) of all components of the active field. Since the parallel **or** reduction can be very efficiently performed in the Connection Machine, we do not execute the main loop of the search by runs. Inside the main loop of the search, only those packets that are still active have to be routed. Therefore, we select only the processors where the active field is true. The last step in the search can be efficiently done using *scan sets* and the scan function.

The implementation of the procedure *route* should also be easy, although three separate reading phases are needed.

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

  /* initializations */
  a→first=−1; a→last =−1;
  [0] (a→first)=0; [0] (a→last)=k−1;
  [0] (a→node)=header(S); [0](a→level)=level(S);
  with (packet_shape) /* only processor nr. 0 is active */
  a→active=(pcoord(0)==0);
  ...
  /* main loop */
  with (packet_shape)
    while (|=a→active)
      where (a→active)
        route (a, S);
  /* spread the current node of each packet to fill 'a→node' /*
  ...
  }
```

The number of stages in a search can be reduced if, whenever a packet is split, we try to route the two resulting subpackets in the same stage. This is possible if the key that hits the packet does not hit any of the two subpackets. The main lemmas about read conflicts, as well as Theorem 3.1, hold true for this variant. Note that in this variant,

any packet p such that remains(p) is true after stage t must be split in stage $t+1$. Although this version of the search guarantees at most the same number of stages as the original algorithm, the execution of the main loop by runs (see Section 3.2) and the increased complexity of each routing steps makes this choice less interesting that it could seem at a first glance.

We can benefit from the use of C*'s **scan** function to implement the insertion and deletion procedures. For instance, it can be used to spread the information collected in the *update* arrays during the initial search phase. It is also useful to efficiently build the skip arrays *succ*, *pred* and *last*, since **scan** is a kind of generalized parallel prefix facility. The *update* and skip arrays used in the insertion and deletion procedures are bidimensonal parallel variables. The number of positions in each rank or dimension can be dynamically declared using **allocate_shape** and then the variables themselves allocated using **palloc**. Note that this simple approach requires $O(k \log n)$ processors (on the average) to storate the *update* array and $O(k \log k)$ processors for each skip array. Since the skip list itself must be stored in the local memories of the processors (not in the host) an expected number of $O(n)$ processors is needed, anyway.

In the case of the insertion, it will be useful to generate the levels of the new nodes and compute its maximum level before allocating the *update* array. In general, the *update* needs $O(k \log n)$ space (on the average) to be stored. But since we know in advance the maximum level m among the new items, only $O(k \log k)$ space will be needed for the *update* array. If $m <$ level(S) then this strategy should be used. The code of the procedure *search_with_update* gets messier, but it is more space efficient.

Finally, we discuss a simple way to save both space and expensive key comparisons. In Section 2 we assumed that every node/level in a skip list stored a copy of the key of the node. That assumption is useful since we have shown that there is at most a constant number of packets at a node/level in any given stage, but it would be interesting to avoid such a space consuming solution. We cannot simply let all packets at the predecessors of a given node access a single copy of the key, because the number of such packets is not bounded by a constant: there can be an expected number of $O(\log n)$ packets trying to look at the single copy of the key of a node during any given stage and, in principle, $O(\log \log n)$ time will be need to execute a stage in an EREW model.

The trick is to keep a pointer already_checked(p) in each packet p. It points to the last node to which the packet "looked at" in the previous current level. The pointer is initialized to **NIL** and updated each time the packet drops one level. The procedure *route* is slightly modified so each routing step begins comparing the pointer already_checked(p) and the forward pointer of the current node/level, say forward(x, l). If they are equal, the packet is pushed down. Otherwise, the key of forward(x, l) is read and the packet is split, pushed forward or down according to the rules given in Section 3. The protocol for exclusive read (see Section 3.1) must be used for both reading the forward pointer and the key of the node it points to. The crucial point is that, if the level of the node forward (x, l) is greater than l, then

already_checked(p) = forward(x, l). Hence, we can keep a single copy of the key of each node, since the keys are exclusively read by packets "looking" through forward pointers that access top node/levels. In other words, a comparison between keys is needed only if a packet is at some node/level (x, l) and level(forward$(x, l)) = l$. Moreover, the key of a node is not to be read more than once by any given packet if we keep a copy of the last read key in each packet. Then, if a packet was generated by a split and was not moved during the same step, the copy of the last read key can be used in the next stage.

The idea of already_checked pointers was suggested by Pugh in [24] as an optimization to reduce the number of key comparisons in the sequential search algorithm. A detailed performance analysis can be found in [16].

7. Conclusions

We have the presented a top-down design of a parallel dictionary based on skip lists, where two points should be emphasized:

(i) The algorithms are easy to explain and justify and do not use recursion. Therefore it seems possible to obtain practical and efficient implementations on real machines without too much effort. The expected behavior of the algorithms is comparable with that of other proposed algorithms and the implementation constant factors are likely to be small.

(ii) They use few processors and a weak model of parallel computation.

The design of these algorithms takes advantage of the fact that skip lists share some of the properties of trees and the simplicity of the linked lists. The main ideas involved in the search algorithm can be summarized as: "route a set of packets on the skip list and split a packet whenever some accessed key hits it". This algorithm reinforces the view of skip lists as trees. The approach for both the insertion and deletion algorithms can be rephrased as: "after the search phase has been done, parallelize the usual updating algorithms for sequential linked lists using extra memory to do fast address arithmetic". Both algorithms dealt with the skip list as a set of independent linked lists once the search phase is finished. An accurate analysis of the search algorithm has pointed out a rich combinatorial structure of the set of packets generated on this algorithm. This structure has been used to give a simple protocol to do exclusive reads, thus allowing an efficient implementation of the algorithms in EREW machines. All these facts lead us to propose the skip lists as one of the practical methods of choice for implementing dictionaries in SIMD machines.

Acknowledgements

We thank Torben Hagerup for the help in this research. We also thank L. Bougé and the anonymous referee for many useful suggestions.

References

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

[2] H. Bast, M. Dietzfelbinger and T. Hagerup, A perfect parallel dictionary, in: I.M. Havel and V. Koubek, eds., *Proc. 17th Math. Foundations Comput. Sci.* Lecture Notes in Computer Science, Vol. 598 (Springer, Berlin, 1992) 133–141.

[3] G.E. Blelloch, Scans as primitive parallel operations, *IEEE Trans. Comput.* **38** (11) (1989) 1526–1538.

[4] L. Bougé, Y. Le Guyadec, G. Ultard and B. Virot, A proof system for a simple data-parallel programming language, in: C. Girault, ed., *Proc. IFIP WG10.3 Internat. Conf. on Appl. in Parallel and Distributed Computing* (Elsevier, Amsterdam, 1994).

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

[6] L. Devroye, A limit theory for random skip lists, *Ann. Appl. Probab.* **2** (3) (1992) 597–609.

[7] M. Dietzfelbinger and F. Meyer aur der Heide, Dynamic hashing in real time, in: J. Buchmann, H. Ganzinger and J.W. Paul, eds., *Informatik. Festschrift zum* **60**. *Geburtstag von Günter Hotz. Teubner-Texte zur Informatik 1* (B.G. Teubner, Leipzig, 1992); also in *Proc. 17th ICALP*, Lecture Notes in Computer Science, Vol. 443 (Springer, Berlin, 1990).

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

[9] T. Duboux, A. Ferreira and M. Gastado, MIMD dictionary machines: from theory to practice, in: L. Bougé, M. Cosnard, Y. Robert and D. Trystram, eds., *Proc. 2nd Joint Internat. Conf. on Vector and Parallel Processing (CONPAR92-VAPPV)* Lecture Notes in Computer Science, Vol. 634 (Springer, Berlin, 1992) 545–550.

[10] J. Gabarró and R. Gavaldà, An approach to correctness of data parallel algorithms, *J. Parallel and Distributed Computing* **22** (1994) 185–201.

[11] J. Gabarró, C. Martínez and X. Messeguer, Parallel update and search in skip lists, in: R. Baeza-Yates, ed., *Computer Science 2: Research and Applications* (Plenum, New York, 1994) 15–26.

[12] M. Gastaldo, Dictionary machine on SIMD archictectures, Tech. Report. 93-19, Laboratoire de l'Informatique du Parallélisme (LIP), 1993.

[13] J. Gil, Y. Matias and U. Vishkin, Towards a theory of nearly constant time parallel algorithms, in: *Proc. 32nd Foundations of Comput. Sci.* (1991) 698–710.

[14] L. Higham and E. Schenk, Maintaining B-trees on an EREW PRAM, *J. Parallel and Distributed Computing* **22** (1994) 329–335.

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

[16] P. Kirschenhofer, C. Martínez and H. Prodinger, Analysis of an optimized search algorithm for skip lists, *Theoret. Comput. Sci.* **144** (1995) 199–220.

[17] P. Kirschenhofer and H. Prodinger, The path length of random skip lists, *Acta Inform.* **31** (8) (1994) 775–792.

[18] C.E. Leiserson, Area-efficient VLSI computation, Ph.D. Thesis, Carnegie-Mellon University, 1981, ACM Doctoral Dissertation Award 1982 (MIT Press, 1983).

[19] X. Messeguer, A sequential and parallel implementation of skip lists, Tech. Report LSI-94-41-R, LSI-UPC, 1994.

[20] T. Papadakis, Skip lists and probabilistic analysis of algorithms, Ph.D. Thesis, University of Waterloo, 1993, available as Tech. Report CS-93-28.

[21] 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, eds., *Proc. 2nd Scandinavian Workshop on Algorithm Theory*, Lecture Notes in Computer Science, Vol. 447 (Springer, Berlin, 1990) 160–172.

[22] W. Paul, U. Vishkin and H. Wagener, Parallel computation on 2–3 trees, in: J. Díaz, ed., *Proc. 10th Internat. Colloquium on Automata, Programming and Languages*, Lecture Notes in Computer Science, Vol. 154 (Springer, Berlin, 1983) 597–609.

[23] W. Paul, U. Vishkin and H. Wagener, Parallel computation on 2–3 trees, *RAIRO Inform. Théor.* (1983) 398–404.

[24] W. Pugh, A skip list cookbook, Tech. Report CS-TR-2286.1, Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, College Park, MD, 1990, also published as UMIACS-TR-89-72.1.
[25] W. Pugh, Skip lists: a probabilistic alternative to balanced trees, *Comm. ACM* **33** (6) (1990) 668–676.
[26] J.T. Schwartz, Ultracomputers, *ACM Trans. Programming Languages Sys.* **2** (4) (1980) 484–521.
[27] S. Sen, Some observations on skip-lists, *Inform. Process Lett.* **39** (3) (1991) 173–176.
[28] Thinking Machines Corporation, Cambridge, MA, C* *Programming Guide. Version 6.0.2*, 1991.