Dynamic Hashing Strategies

Friedhelm Meyer auf der Heide [1]
Mathematik-Informatik
Universität-GH Paderborn
4790 Paderborn
Fed. Rep. of Germany

Abstract

This survey paper describes new types of dynamic hashing strategies for implementing dictionaries on sequential, parallel and distributed computation models. In particular, it surveys the progress in constructing and analyzing new classes of universal hash functions.

1 Introduction

Dynamic hashing strategies are a well known method to implement dictionaries. A dictionary is one of the most basic, most important data structures. It supports the operations Insert, Delete, Lookup. More precisely, a dictionary contains data items that are identified by keys from a universe U. We shall assume that $U = I\!N$ or $U = \{1, \ldots, p\}$ for some integer p. A data item is a pair (x, Info), where $x \in U$ is the key and Info is the information associated with key x (for example, x is a name of a person and info is a record contain information about this person). If S is the set of keys associated to the data items currently stored in the dictionary, then

- Insert(x, Info) adds (x, Info) to the dictionary and deletes (x, Info') from the dictionary, if $x \in S$.
- Delete(x) deletes the data item with key x from the dictionary, if $x \in S$.
- Lookup(x) returns the data item with key x, if $x \in S$, and a default message else.

In what follows we shall, for ease of description, surpress the information associated to a key. Thus a dictionary contains just a set S of keys from the universe U.

As the dictionary is one of the most important data structures, many implementations were proposed, for an overview see [Me 84]. The most important of them belong to one of the following two categories:

Pointer-based strategies
Here pointers are mainly used to implement variants of dynamic search trees as AVL-trees, (a, b)-trees, finger-trees etc.

Dynamic hashing strategies
Here a hash function h is used that maps the universe into a hash table i.e. an array, whose size m should not be much larger than the size n of the set S of currently stored keys. There are several strategies known to handle collisions, i.e. cases where different keys are mapped to the same table position, see [Me 84]. We shall only consider the simple variant "hashing with chaining".

[1] supported in part by DFG grant ME 872/1-3

The keys of S that are mapped to the same table position i form the i'th bucket B_i (of S under h). "Hashing with chaining" means that each table position is a linked list that stores the corresponding bucket.

For lookup(x), one has to check whether bucket $B_{h(x)}$ contains x. For Insert(x) or Delete(x) one has to execute lookup(x) and then modify $B_{h(x)}$ accordingly.

In order to need only little storage, the table size m should not be much larger than the size n of the current dictionary.

Further, as the size of the dictionary changes dynamically by insertions and deletions, the whole data structure has to be rebuilt from time to time.

Thus, the performance of a hashing strategy depends on

- space, i. e. the size of the hash table,
- reconstruction time, i. e. the time needed to construct a new hash function, to set up the corresponding hash table, and to insert the keys from the old dictionary,
- evaluation time, i. e. the time needed to evaluate the hash function,
- bucket sizes, (we shall see several measures for "the buckets are small").

We distinguish between four types of hashing strategies.

Uniform hashing

Here one considers deterministic algorithms using a fixed hash function. A "good" hash function has to distribute the universe U evenly among the positions of the hash table, as e. g. $x \mapsto x \bmod m$ does. The dictionary is reconstructed whenever the size of the dictionary doubles or halfes. If one chooses m only a little larger than n, it is well known that one achieves $O(n)$ space (optimal), $O(n)$ reconstruction time (optimal) and constant evaluation time (optimal) with the above hash function. It is not hard to show (see e. g. [Me 84]) that the average time to execute n instructions is $O(n)$ (optimal), if one averages over all sequences of n instructions (lookup(x_1), Insert(x_2)...).

Average time is a realistic complexity measure only if the keys used in the sequence of instructions are close to a random sequence. This is usually not the case. Therefore, Carter and Wegman introduced in [CW 77] a new class of hashing strategies called

Universal Hashing

Here one applies a probabilistic algorithm that randomly chooses a hash function from a *universal class H of hash functions*. *Universal* means that for a given set S of keys a randomly chosen $h \in H$ yields fast execution of a sequence of operations in which the keys from S are involved. The runtime here is measured as worst case runtime over all sequences of n instructions.

For fixed such S, one considers the expected runtime, or, even more, demands that the runtime be small with high probability, where expectation and probabilities are relative to the random choice of the hash function from H. Reconstruction is executed in the same situations as described for uniform hashing, and, in addition, if the currently used hash function turns out to perform badly.

It is well known (see e. g. [Me 84]) that one can achieve linear space and expected linear time for such dynamic universal hashing strategies, if H is chosen e. g. as $H = \{h_{a,b} : U \to \{1, \dots, m\},$

$h_{a,b}(x) = (ax + b) \bmod p) \bmod m$, for $a, b \in U\}$, where $U = \{1, \ldots, p\}$, p prime. (More on this and other universal classes of hash functions is shown in the next chapter.)

It is particularly important that lookups are executed fast, because a lookup instruction demands an answer. The time to perform a lookup is called response time.

Perfect hashing
stands for universal hashing strategies that guarantee worst case constant response time. For this purpose one needs classes of perfect hashing schemes, usually composed of universal hash functions. Fredman, Komlós and Szemerédi ([FKS 84]) have presented such a class of perfect hashing schemes with good performance (linear space, linear expected reconstruction time). For perfect hashing, the step from a static dictionary (i. e. one where the structure is built up once for a fixed set S, then only lookups are supported) to a (dynamic) dictionary is more difficult than in the uniform or universal case, because the property "worst case constant lookup time" demands much more sopisticated strategies to handle insertions. Such a dynamic perfect hashing strategy with expected optimal time and space requirements is developed by Dietzfelbinger et al in [DK* 88].

An "ideal" dictionary should have the property, that each instruction is executed in constant time.

Monte Carlo type hashing
comes very close to this ideal situation. Here one demands that each instruction takes worst case constant time. One still uses a probabilistic algorithm. The Monte Carlo property means that one allows a (very small) probability that the algorithm fails to fulfil the runtime assumption.

Very recently, Dietzfelbinger and the author have developed such a strategy in [DM 90]. It uses linear space and the probability of failure is polynomially small in n, if $\frac{n}{2}$ instructions are executed on a dictionary of initial size n.

For universal, perfect, and Monte Carlo type hashing, one needs classes of universal hash functions with high performance.

In the next chapter we shall survey the recent suggestions for universal classes of hash functions and their properties (see [CW 77], [WC 79], [FKS 84], [DK* 88], [Sie 89], [DM 90]). In the third chapter we sketch the static perfect hashing scheme from [FKS 84] and its dynamisation from [DK* 88]. Chapter 4 sketches the idea of the Monte Carlo Type hashing strategy form [DM 90]. In chapter 5 we survey recent implementations of hashing strategies on parallel machines with and without shared memory ([DM 89], [DMa 90]), and discuss applications to emulations of shared memory.

2 Universal Classes of Hash Functions

We start by describing several ways of measuring the goodness of a class of hash functions $H \subseteq \{h : U \to \{1, \ldots, s\}\}$. The classical notion of (c, k)-universality is introduced by Wegman and Carter. The idea behind this notion is that H is well suitable for a universal hashing strategy, if the probability of collisions is small. Probabilities are relative to the random choice of $h \in H$.

1) H is *(c, k)-universal*, if for each $j \leq k$, $x_1 < \ldots < x_j \in U$, it holds that $\text{Prob}(h(x_1) = \ldots = h(x_j)) \leq \frac{c}{n^{k-1}}$.

The following notions describe several ways of formalizing the idea that bucket sizes should be small. (Assume a set $S \subseteq U$, $B_1, \dots, B_s$ are the buckets S is split into by a randomly chosen $h \in H$, $b_1, \dots, b_s$ are their sizes (compare introduction).)

2) **maximum bucket size:** $\max\{b_i, i = 1, \dots s\}$ should be small with high probability, or at least expected.

3) **individual bucket size:** For each $i = 1, \dots, s$, b_i should be small with high probability, or at least expected.

4) **f-weighted bucket sizes:** Consider a function $f : I\!N^s \rightarrow I\!N$. $f(b_1^h, \dots, b_s^h)$ should be small with high probability, or at least expected. For example, the key lemma in the analysis of the perfect hashing scheme of [FKS 84] shows that for their class H of hash functions, if $s = n$ then $E(\sum_{i=1}^{n} b_i^2) = O(n)$ (see below).

   Further important criteria for the goodness of a universal class H are

5) **evaluation time,**

6) **reconstruction time and space,**

   as described in the introduction. Now let us analyse a well known class of hash functions.

Polynomials as hash functions
We assume from now on that $U = \{1, \dots, p\}$ for a prime number p. Let $H_s^d := \{h_a : U \rightarrow \{1, \dots, s\}, a = (a_0, \dots, a_d) \in U^{d+1}\}$ where $h_a(x) := (\sum_{i=0}^{d} a_i\, x^i \bmod p) \bmod s$.

This class is considered and analyzed in several papers as we shall see. We shall consider d a constant. The following list L1 shows properties of H_s^d. Let $S \subseteq U$, $\#S = n$, be fixed.

List L1: (The numbering refers to the numbering of properties of hash functions from above.)

1) H_s^d is $(1 + o(1), d + 1)$-universal ([WC 79]).

From now on we assume that h is randomly chosen from H_n^d.

2) $\mathrm{Prob}(\max\{b_i, 1 \le i \le n\} \le c \cdot n^{1/d+1} \ge 1 - O(c^{-d})$ (can be concluded from [DK* 88]).

3) For $i \in \{1, \dots, n\}$, $\mathrm{Prob}(b_i \le u) \ge 1 - 0(u^{-d})$ (can be concluded using results from [DK* 88]).

4) $\mathrm{Prob}(\sum_{i=1}^{n} b_i^{d+1} = O(n)) \ge \frac{1}{2}$ ([DK* 88]).

The results 3) and 5) are also shown in [FKS 84] for $d = 1$. Some of the results can be concluded from 1) as shown in [MV 84] and [WC 79].

Obviously, also the following holds:

5) Evaluating $h \in H_s^d$ needs time $O(d) = O(1)$.

6) Reconstruction, i.e. choosing a random $h \in H_s^d$ and inserting n elements in the corresponding hash table needs space and time $O(d + s + n) = O(s + n)$. (This even holds for space and expected time if $s = n$ and the property from 4) is demanded.)

We shall see that polynomials as hash functions are well suited for dynamic perfect hashing on sequential or parallel machines with shared memory. But for Monte Carlo type hashing or for parallel hashing without shared memory, it seems that one needs hash functions that are closer to random functions.

"Almost random" hash functions.

The following class of hash functions is introduced, analyzed, and applied in [DM 90] and [DMa 90]. It is composed of polynomials. Let again $U := \{1,\ldots,p\}$, p prime. For $r, s \geq 1$ and $d \geq 1$ we define $R(r,s,d) := \{h : U \rightarrow \{1,\ldots,s\}$, $h = h(f,g,a_1,\ldots,a_r)$ for some $h \in H_r^d$, $g \in H_s^d$, $a_1,\ldots,a_r \in \{1,\ldots,s\}\}$, where $h = h(f,g,a_1,\ldots,a_r)$ is defined by $h(x) = (g(x) + a_{f(x)}) \bmod s$.

We shall see that this class has constant evaluation time, linear reconstruction time but shares many properties of truly random functions, as can be seen from the list L2 below.

For ease of description, we fix n, a set $S \subseteq U$ with $\#S = n$, and d. We further assume that $r = n^{1-\delta}$, $s = n$ for some $0 < \delta < 1$, and write R for $R(n^{1-\delta}, n, d)$. All results in this chapter are from [DM 90] and [DMa 90]. First we single out a class $R(S) \subseteq R$. We shall see that a random $h \in R(S)$ will behave almost like a random function on S, and that a random function from R belongs to $R(S)$ with high probability.

A function $h : U \rightarrow \{1,\ldots,s\}$ is *l-perfect* on S, if all bucket sizes are at most l.

Lemma 2.1: Let $R(S) := \{h = h(f,g,a_1,\ldots,a_r) \in R \mid$ with $b_i^f \leq 2n/r$ and g is d-perfect on B_i^f for $i = 1,\ldots,r\}$. Then for a randomly chosen $h \in R$ it holds that $\text{Prob}(h \in R(S)) \geq 1 - O(n^{-k})$, where k can be made arbitrarily large by choosing sufficiently large d.

List L2: (The numbering refers to the numbering of properties of hash functions from above.)

1) R is $(1 + o(1), d + 1)$-universal. (this certainly is far away from random functions, which are $(1 + o(1), \infty)$-universal.)

   Form now on we assume that h is randomly chosen from $R(S)$. In order to get probability bounds for a randomly chosen $h \in R$, we have to multiply the probability bounds below by $1 - O(n^{-k})$, see Lemma 2.1. To make clear that h is chosen from $R(S)$ we write $\text{Prob}_S(\ldots)$ for $\text{Prob}(\ldots \mid h \in R(S))$, and $E_S(\ldots)$ for $E(\ldots \mid h \in R(S))$.

2)a) $\text{Prob}_S(\max\{b_i \mid 1 \leq i \leq n\} \leq u) \geq 1 - 0(u^{-u/2d})$ for $u = \Omega(\log(n)/\log\log(n))$.

b) $E_S(\max\{b_i \mid 1 \leq i \leq n\}) = O(\log(n)/\log\log(n))$.

3) For $i = 1,\ldots,n$, $\text{Prob}_S(b_i \leq u) \geq 1 - (e^{u-1}/u^u)^{1/d}$.

4) $\text{Prob}_S(\sum_{i=1}^{h} b_i^{(1-\varepsilon)b_i} = O(n)) \geq \frac{1}{2}$ for sufficiently large $\varepsilon, 0 < \varepsilon < 1$, dependent on d.

All the above probability bounds are the same as for random functions, up to a factor only dependent on d in the exponents. The crucial property of $R(S)$ is 3) which almost trivially includes 2) and 4). Further, it is easily seen that

5) Evaluation of $h \in R$ needs time $O(d_1 + d_2) = O(1)$.

6) Reconstruction needs space and time $O(n)$, if a random $h \in R$ is used. If the set S has to be reconstructed, space $O(n)$ and expected time $O(n)$ can still be achieved when a random $h \in R(S)$ shall be used.

For sake of completeness we here note further progress in the classical approach of defining

Further (c,k)-universal classes of hash functions.
The result 1) of list L1 also holds for polynomials whose degree grows with n. Specifically, results 2), 3), 4), 6) for $R(S)$ from L2 also hold for $H_n^{\log(n)}$ in a similar way, but this class has the drawback that the evaluation time is $\Theta(\log(n))$, not constant. $H_q^{\log(q)}$ is used in shared memory emulations on networks to distribute the universe of shared memory addresses among the q processors of the network (see [MV 84], [U 84], [R 87], [MV 84], [KU 86].) We shall discuss in chapter 5 how the new hash functions from $R(q^{1-\delta}, q, d)$ can be applied here.

A different approach to construct (c,k)-universal classes for large k is shown by Siegel in [Sie 89].

He combines randomly chosen bipartite graphs of fixed degree and polynomials. As almost all bipartite graphs of fixed degree have a certain "concentrator property" Siegel can show that his construction yields a (c, n^{ε})-universal class of hash function, $\varepsilon > 0$ can be chosen arbitrary. From this universality, features similar to 2), 3), 4) from L2 can be shown, as done in [WC 79] and [MV 84]. Siegel's functions need linear space and reconstruction time, and constant evaluation time, if the universe has size n^k for some constant k. A drawback is that both evaluation time and reconstruction time and space contain a factor exponential in k.

3 Perfect hashing strategies

We start with a very simple, time efficient perfect hashing scheme which needs superlinear space. Its construction and analysis is implicit in the following lemma. Again we assume that $U = \{1, \ldots, p\}$, p prime.

Lemma 3.1 ([DK* 88]) Let $\varepsilon > 0$, $k \geq 1$, $d = d(\varepsilon, k)$ sufficiently large, $S \subseteq U$, $\#S = n$. Let h be randomly chosen from H_s^d with $s = n^{1-\varepsilon}$. Then Prob(h is d-perfect on S) $\geq 1 - O(n^{-k})$. For $d = 1$ and $\varepsilon = 1$ one achieves a probability bound of $\frac{1}{2}$, as shown already in [FKS 84].

Fredman, Komlós, and Szeméredi have combined this lemma for $d = 1$ and property 4 from list L1. They use the following hashing scheme. Let $S \subseteq U$, $\#S = n$ be fixed.

1) Construct a randomly chosen $h \in H_n^1$, the *primary hash function*. Test whether it splits S into buckets $B_1, \ldots, B_n$ satisfying $\sum b_i^2 = c \cdot n$ (for some small constant c). If not, try again.

2) For $i \in 1, \ldots, n$, construct randomly chosen $h_i \in H_{s_i}^1$, the *secondary hash functions*, with $s_i = c' b_i^2$, (for some small constant c') and check whether it is perfect (i.e. 1-perfect) on B_i. If not, try again.

3) Use a header table HT of length n and a secondary table ST of length $c \cdot c' \cdot n$. Store a description of h_i and $l_i := \sum_{j=1}^{i-1} c' b_j^2$, $l_1 = 0$, in HT[i].

4) Store $x \in S$ in $ST[h_{h(x)} + l_{h(x)}]$.

Clearly, a lookup now costs 0(1) time. ($x \in S \Longleftrightarrow ST[h_{h(x)} + l_{h(x)}] = x$).

By property 4 from L1 and Lemma 3.1, the expected time for constructing the scheme is $O(n)$. The space needed is $O(n)$ in the worst case. More precisely it can be shown.

Theorem 3.2 [FKS 84]. The above scheme needs $O(n)$ space, constant lookup time. Further, Prob(time to construct the scheme $> L \cdot c \cdot n) < 2^{-L}$ for some constant c and all $L \geq 1$.

Now assume that we want to execute n instructions on an initially empty dictionary, using $O(n)$ space.

We start by setting up a hashing scheme as above with ST having length $c \cdot c' \cdot n$. We choose a random $h \in H_n^2$, but not yet the h_i's. We define all $B_i := \emptyset$.

In the simplest version, we do the following: Whenever an instruction with key x is read do the following:

- compute $i := h(x)$.

For a lookup: Test whether h_i is already defined (i.e. $B_i \neq \emptyset$) and if yes whether $h_i(x) = x$.

For a deletion: First lookup x. If it is found then mark it with a tag "deleted".

For an insertion: First lookup x. If it is found with tag "deleted" then remove the tag. If it is found without tag, do nothing. If it is not found, compute $i := h(x)$. Let $B_i := B_i \cup \{x\}$. Use a new segment $ST[a+1], \ldots, ST[a+s]$ of ST, where a the largest address of a previously used position of ST, and $s := c' b_i^2$. Choose randomly $h_i \in H_s^1$ until one is found which is perfect on B_i. Store x in $ST[a + h_i(x)]$. Change $HT[i]$ appropriately.

If ST is exhausted before the n instructions are executed, reconstruct the hashing scheme.

The main observations for the analysis are as follows:

- Space is $O(n)$ by definition of the sizes of HT and ST.
- Lookup time is worst case constant.
- Let S be the set of keys used in the n instructions, then $\#S \leq n$. Then, if $h \in H_n^2$ is chosen such that $\sum_{i=1}^n b_i^3 = c \cdot n$, no reconstruction is necessary, because only space $\sum_{i=1}^n (c' \cdot 1^2 + c' \cdot 2^2 + \ldots c' \cdot b_i^2) \leq c \sum_{i=1}^n b_i^3 \leq c \cdot c' \cdot n$ is used in ST. But this is the length of ST.
- Similarly, if h fulfills $\sum_{i=1}^n b_i^3 \leq c \cdot n$, the expected time needed is $O(n)$, because by Lemma 3.1, the expected number of attempts until an h_i is found (see "for an insertion") which is perfect on B_i is at most 2.
- By property 4 from L1, $\sum_{i=1}^n b_i^3 \leq cn$ is fulfilled by h with probability $\frac{1}{2}$. Thus the expected number of reconstructions is at most 2.

Altogether we get a linear expected time bound. In order to adapt the space needed to the varying size of the dictionary, reconstructions are also executed in situations as described for uniform hashing in the Introduction. One easily verifies that this does not damage the time and space bounds. Further improvements can be done:

- It is possible to choose the primary hash function h from H_n^1.
- If one chooses the primary hash function h from H_n^d, one can show, using property 3 from L1, that Prob(Instruction j needs more than u steps)$=O(u^{-d})$, which extends the above expected

constant time bound. Altogether, with the help of polynomials as hash functions, one can achieve:

Theorem 3.2 [DK* 88]. There is a dynamic perfect hashing strategy with worst case constant lookup time, space requirement linear in the current size of the dictionary, and Prob(instruction j needs more than u steps) $= O(u^{-d})$ for each u. The dictionary can be built for arbitrary constants d. The expected time for n instructions executed on an initially empty dictionary is $O(n)$.

4 Monte Carlo type hashing strategies

For such strategies with linear space bound, we need the "almost random hash" functions described in Section 2. We start with a dictionary of the Monte Carlo type which uses superlinear space. It can be easily derived from Lemma 3.1.

Theorem 4.1 ([DM 90]). Let $\varepsilon > 0$, $k = k(\varepsilon)$ be fixed. There is a Monte Carlo Type dictionary that, with probability exceeding $1 - O(n^{-k})$, executes n^{ε} instructions in such a way that each of them takes constant time in the worst case, if space $O(n)$ is available.

In order to construct a dictionary as above with *linear* space requirement, we first show how to construct a dictionary restricted as below:

A type-A dictionary is a data structure with the following properties. Consider a sequence $I_1, \ldots, I_n$ of n insertions and deletions. Fix $\varepsilon > 0$, $k \geq 1$.

a) Space $O(n)$ is used.

b) The operations are performed in $n^{1-\varepsilon}$ phases. In phase l the instructions $I_{(l-1)n^{\varepsilon}+1} \ldots I_{ln^{\varepsilon}}$ are performed. Each phase takes time $O(n^{\varepsilon})$.

c) A lookup request asked during phase l needs constant time and returns the correct answer if the key asked for has not yet occured during phase l.

Theorem 4.2. Let $\varepsilon > 0, k \geq 1$ be fixed. There is a type-A dictionary of Monte Carlo type that fails to fulfil (b) with probability $O(n^{-k})$.

The algorithm proceeds as follows. Let $S = \{x_1, \ldots, x_n\}$ be the set of keys used in the first n instructions, $S(l)$ those keys used in phase l.

Choose $h \in R$ (compare Sektion 2, "almost random" hash functions) at random. h splits S into buckets $B_1, \ldots, B_n$. B_i is stored in a static dictionary as described in Theorem 3.2 as follows:

in Phase l: Evaluate $h(x)$ for all $x \in S(l)$. Collect $S(l) \cap B_j$ in a list T_j^2. Let $J := \{j, T_j^2 \neq \emptyset\}$. For each $j \in J$ construct a list T_j^1 of all keys from B_j already stored in previous phases. Concatenate T_j^1 and T_j^2 to a list T_j, for all $j \in J$. Construct new subdictionaries for all T_j, $j \in J$, as described in Theorem 3.2.

We only sketch some ideas for the analysis. The time needed in phase l is essentially described by $\sum_{j \in J} |T_j|$. Thus we need that this quantity is $O(n^{\varepsilon})$ with high probability. The (fairly involved) key lemma we need is as follows.

Lemma 4.3 [DM 90]. With the notions from the above algorithm: Prob_S(more than $4dn^\epsilon$ keys from S belong to a bucket B_j with $j \in J) = O(n^{-k})$. $k = k(\delta, d)$ can be made arbitrarily large. (For δ, d, and $\mathrm{Prob}_S(\ldots)$ compare the definitions of the classes R and $R(S)$ from Section 2.)

As constructing the hashing scheme from Theorem 3.2 for one T_j has only constant probability to be fast, we need a further probability bound which says that under these circumstances, the probability is high that constructing the hashing schemes for all T_j's, $j \in J$, is fast, i. e.we need a good bound for the probability of bounds of the sum of independent, exponentially distributed random variables with certain properties. This can be found in [DMa 90]. The arguments above suffice to prove Theorem 4.2.

In order to get a Monte Carlo type dictionary we now combine Theorems 4.1 and 4.2. We use the notions of the algorithm for Theorem 4.2.

in Phase l, we concurrently, in an interleaving fashion, do two things:

- Execute the insertions and deletions of phase l in a dictionary D_l as described in Theorem 4.1.
- Insert the keys from D_{l-1} into a background dictionary as described in Theorem 4.2.
- A lookup of phase l is executed in D_l, D_{l-1}, and the backup dictionary in this order, until the key is found. If it is found in none of the dictionaries, it is not stored at all.

It is not too hard to implement and analyse (using Theorems 4.1 and 4.2) the above algorithm to show that n instructions can be executed on an initially empty dictionary in such a way that, with high probability, each instruction needs constant time.

Adapting the strategy to the varying size of the dictionary now needs some more effort, because we are not allowed to interrupt the stream of instructions for a while. For this a global backup dictionary is necessary, i. e.we need three concurrently working dictionaries altogether. We then obtain the result:

Theorem 4.4 [DM 90]. There is a Monte Carlo Type dictionary with worst case linear space requirement and worst case constant lookup time. Further, with probability $O(n^{-k})$ (k be the chosen arbitrarily), $\frac{1}{2}n$ instructions on a dictionary of initial size n can be executed in such a way that each of them needs constant time.

5 Parallel and distributed dictionaries, with applications to shared memory emulations

Assume a synchronized parallel machine with q processors, where each processor gets instructions (Lookup, Insert, Delete) from its (virtual) user. All the users manipulate and lookup in a (virtually) shared dictionary. In [PVW 83], 2-3 trees are implemented on a parallel random access machine (PRAM), i. e. a parallel machine with random access to a shared memory. As in the sequential case, $O(\log(n))$ time is necessary for an instruction on a dictionary of size n. Thus, n instructions, $\frac{n}{q}$ from each user, need $O(\frac{n}{q}\log(n))$ time to be executed.

In [DM 89] a parallel perfect hashing strategy is implemented on a PRAM. Polynomials as hash functions are used, and a variant of the hashing strategy from [DK* 88] is developed. One needs much better probability bounds than in the sequential case, because one typically has to consider the expected time of the *slowest* processor. Therefore, linear hash functions do no longer suffice, and it seems that it is a bad idea to let every processor execute its own instructions. Instead, the instructions are distributed randomly among the processors. In [DM 89] the following is shown.

Theorem 5.1 ([DM 89]). There is a parallel perfect hashing strategy on a PRAM with q processors with worst case linear space requirement and worst case constant lookup time. $n \geq q^{1+\varepsilon}(\varepsilon > 0$ can be made arbitrarily small) instructions, $\frac{n}{q}$ per processor, can be executed in expected time $O(\frac{n}{q})$.

In [DMa 90], a distributed version of a parallel dictionary, implemented on a complete network of processors without shared memory is presented. The basic idea is as follows:

Assume a functional subdivision of a processor in three units, the input-, working-, and dictionary processor (IP, WP, DP).

In a round, each IP chooses a random WP. If the WP is currently idle (i. e. not busy with executing an instruction) and only one IP has chosen this WP, then the IP asks its user for a new instruction and forwards it to the WP.

The WPs execute instructions as follows. They have initially chosen a distribution hash function DH from $R(q, q^{1-\delta}, d)$ (compare Section 2). A key x is processed and stored in the k-th DP, with $k := DH(x)$.

The WP tries to pass its instruction to this DP. The main problem arising here comes from the collisions at the DP, because many WPs may want to pass an instruction to the same DP. The algorithmically most involved part of the strategy is the handling of these collisions, in particular if many instructions concerning the same key have to be executed concurrently.

In the DPs Monte Carlo Type dictionaries as described in Theorem 4.2 are used. They are good enough to guarantee that the expected runtime even of the slowest of q concurrently working dictionaries for $\frac{n}{q}$ instructions, each, is still $O(\frac{n}{q})$. The dictionary based on polynomials as hash functions from Theorem 3.2 would only guarantee a time bound $O(\frac{n}{q}\log(q))$.

The analysis is fairly involved and omitted in this survey paper. The result is:

Theorem 5.2 ([DMa 90]). A distributed dictionary can be implemented on a complete network of q processors such that $n \geq q^{1+\epsilon}$ instructions, $\frac{n}{q}$ per processor, can be executed in expected time $O(n)$. Space $O(\frac{n}{q})$ per processor is used in the worst case, if n keys are stored. Every lookup needs expected constant time. q concurrently processed lookups are all answered after expected $O(\log(q)/\log\log(q))$ time. $k \geq \log(q)$ lookups from each processor can be answered in expected time $O(\log(q))$.

The analysis extensively uses that DH is chosen from the class R of "almost random" hash functions. In particular, property 3 from list L2 is applied. The two last time bounds from the Theorem also mirror properties of the class R, e. g. the $O(\log(q)/\log\log(q))$ bound mirrors property 2b) from list L2.

Emulations of shared memory

There are several suggestions known how to simulate a PRAM on a network, i. e. how to emulate shared memory.

Closely related to the above result are the shared memory emulations in [U 84], [KU 86], [R 87], [MV 84], [KRS 90].

The first three papers and parts of the fourth apply polynomials of degree $\approx \log(q)$ as distribution hash functions. Thus, these techniques yield "best case" time bounds $\Omega(\log(q))$ for the response time of lookups, and $\Omega(\frac{n\log(q)}{q})$ for executing n instructions even on complete networks of q processors.

The above distributed dictionary can be used to obtain an $O(\log(q)/\log\log(q))$ time bound for simulating a memory access step of a PRAM on a complete network of the same size.

In [KRS 90], it is shown how to simulate a PRAM with q processors on a complete network of $q^{1-\epsilon}$ processors with optimal delay $O(q^{\epsilon})$. They use polynomials of constant degree as distribution hash functions. Combining their techniques with the sequential Monte Carlo type dictionary yields $O(\frac{n}{q})$ expected time for executing n instructions. But as q instructions have to be executed en bloc to obtain the above time bound, only expected $\Theta(q^{\epsilon})$ response time for lookups can be reached.

It is easy to conclude from Theorem 5.2 that a simulation as above can already be achieved when the network even has $q/\log(q)$ processors. The delay becomes $O(\log(q))$.

6 References

[Me 84] K. Mehlhorn, Data structures and algorithms, Vol. 1, Springer Verlag, 1984.

[FKS 84] M. L. Fredman, J. Komlós, E. Szemerédi, Storing a sparse table with $O(1)$ worst case access time, J. ACM 31(3), pp. 538-544, 1984.

[DK* 88] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, R. E. Tarjan, Dynamic perfect hashing: upper and lower bounds, Proc. of 29th IEEE FOCS, pp. 524-531, 1988.

[DM 90] M. Dietzfelbinger, F. Meyer auf der Heide: A new universal class of hash functions, and dynamic hashing in real time, Proc. of 17th ICALP, 1990.

[DMa 90] M. Dietzfelbinger, F. Meyer auf der Heide: How to distribute a dictionary in a complete network, Proc. of 22nd ACM STOC, 1990.

[DM 89] M. Dietzfelbinger, F. Meyer auf der Heide: An optimal parallel dictionary, Proc. of ACM SPAA, pp. 360-368, 1989.

[CW 77] J. L. Carter, M. N. Wegman, Universal classes of hash functions, Proc. of 9th ACM STOC, pp. 106-112, 1977.

[WC 79] M. N. Wegman, J. L. Carter, New classes and applications of hash functions, 20th IEEE FOCS, pp. 175-182, 1979.

[Sie 89] A. Siegel, On universal classes of fast high performance hash functions, their time-space trade-off, and their applications, Proc. of 30th IEEE FOCS, pp. 20-25, 1989.

[KRS 90] C. P. Kruskal, M. Snir, L. Rudolph, A complexity theory of efficient parallel algorithms, Proc. of 15th ICALP pp. 333-346, 1988; also: revised preprint.

[MV 84] K. Mehlhorn, U. Vishkin, Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memory, Acta Informatica 21, pp. 339-374, 1984.

[U 84] E. Upfal, Efficient schemes for parallel communication, J. ACM 31(3), pp. 507-517, 1984.

[R 87] A. G. Ranade, How to emulate shared memory, Proc. of 28 IEEE FOCS, pp. 185–194, 1987.

[KU 86] A. Karlin, E. Upfal, Parallel hashing, an efficient implementation of shared memory, Proc. of 18th ACM STOC, pp. 160–168, 1986.

[PVW] W. Paul, U. Vishkin, H. Wagener, Parallel dictionaries on 2–3 trees, Proc. of 10th ICALP, pp. 597–609, 1983.