Fast Hashing on a PRAM - Designing by Expectation

by

Joseph Gil
and
Yossi Matias

Technical Report 91-13
June, 1991

Department of Computer Science
University of British Columbia
Vancouver, B.C.
CANADA V6T 1Z2

Fast Hashing on a PRAM

- Designing by Expectation

Joseph Gil *
University of British Columbia †

Yossi Matias
‡Tel-Aviv University and University of Maryland

Januray 1991

Abstract

A *hash table* is a data structure for representing a set of n items that uses linear space and supports membership queries in $O(1)$ time. We show how to construct a hash table for any given set in $O(\log\log n)$ expected time using n processors on a weak version of a CRCW PRAM. We also show that optimal speed-up can be obtained, i.e., the expected running time is n/p, when the number of processors p is $\leq n/\log\log n \log^* n$.

A general paradigm for the design of randomized algorithms is introduced. This method is used in our algorithm and may be useful in the design of other fast randomized algorithms.

1 Introduction.

1.1 Hash Functions and Static Hash Tables.

Let S be a set of n keys drawn from a finite universe U. Let h be a function $U \stackrel{h}{\mapsto} [0,\ldots,s-1]$ (such functions are called *hash* functions). We call h a *perfect hash function* for S if there are no two keys in S that are mapped by h to the same value. We call h a *good lookup function for* S if: *(i)* h is perfect for S; *(ii)* h uses linear storage (i.e., $s = O(n)$); *(iii)* h can be represented in $O(n)$ space; and *(iv)* h supports quick lookups, i.e., for every $x \in U$, $h(x)$ can be evaluated in $O(1)$ time by a single processor. A good lookup function induces an $O(n)$ storage data structure for representing S, with $O(1)$ time for lookup query. Such a data structure is called *linear static hash table.*

*Research supported in part by the Leibniz Center for Research in Computer Science, Jerusalem, Israel.

†Part of this research was done when the author was at the Hebrew University

‡Research supported in part by NSF grant NSF-CCR-8906949

In this paper the problem of constructing in parallel a good lookup function for a given input set is considered. We provide fast and efficient parallel algorithms for this problem.

The algorithms make use of novel techniques that avoid counting and sorting and thereby circumvent the barrier caused by the $\Omega(\log n/\log\log n)$ lower bound for these problems [4]. In particular we demonstrate that hashing is easier than sorting from the parallel perspective, in the sense that it can be done much faster (similarly to the sequential case). The design of algorithms whose fast running time precludes usage of basic tools like counting is especially challenging; this research presents two tools that may be useful in this context: a) A general paradigm for designing fast parallel randomized algorithms. b) A scheme for the use of the load balancing algorithm presented in [16] to achieve optimal speed-up. By this scheme, the load balancing algorithm is used only $O(\log^* n)$ times.

1.2 Previous Work.

In their seminal paper [14], Fredman, Komlós and Szemerédi introduced a sequential scheme that generates a linear static hash table in $O(n)$ expected time for any input set. Their scheme builds a *2-level* hash function: a *level-1* function splits S into subsets whose sizes are distributed in a favorable way. Then, a perfect *level-2* hash function is built for each subset. Dietzfelbinger, Karlin, Mehlhorn, Meyer auf der Heide, Rohnert and Tarjan [10] extended the *2-level* scheme for a dynamically changing input set.

Dietzfelbinger and Meyer auf der Heide [11] introduced a dynamic data structure (dictionary) that can be implemented in parallel. It preserves optimal speed-up, but the time bounds are of the form $O(n^\epsilon)$.

Matias and Vishkin [26] presented a parallel static hashing scheme that takes $O(\log n)$ expected time and preserves optimal speed-up, i.e., it uses $O(n/\log n)$ processors. This result is the fastest PRAM-algorithm previously known for hashing. It is based on the 2-level scheme of [14] and makes extensive use of counting and sorting procedures. They also suggested and demonstrated the applicability of hashing as an important building block for many parallel algorithms.

The only known lower bounds for parallel hashing were given by Gil, Meyer auf der Heide and Wigderson [20]. In their (rather general) model of computation $\Omega(\log^* n)$ is the bare minimum. However, when processors cannot move between keys, $\Omega(\log\log n)$ time is required to obtain $O(1)$ lookup time. Our algorithms fit this model.

1.3 Results.

1.3.1 Parallel Hashing.

Our main result is that a linear static hash table can be constructed in doubly logarithmic time.

THEOREM 1. *A good lookup function can be built in $O(\log\log n)$ expected time and $O(n)$ space, using n processors on a CRCW PRAM.*

This is the best possible result that does not use processors reallocations, as shown in [20]. Optimal speed-up can be achieved with a small penalty in execution time. It is a significant improvement over the $O(\log n)$ time algorithm of [26].

THEOREM 2. *A good lookup function can be built in $O(\log\log n \log^* n)$ expected time and $O(n)$ space, using $n/\log\log n \log^* n$ processors on a CRCW PRAM (optimal speed-up).*

The model of computation is the *collision$^+$-CRCW* PRAM [9] (which is slightly stronger than the standard common-CRCW but weaker than the arbitrary-CRCW), in which if more than one processor attempts to write different values simultaneously into the same cell then a special collision symbol is written in that cell.

Remark. The hashing result demonstrates the power of randomness in parallel computation on CRCW machines with memory restricted to linear size. Boppana [6] considered the problem of Element Distinctness: given n integers decide whether or not they are all distinct. He showed that solving Element Distinctness on an n-processor priority-CRCW machine with bounded memory requires $\Omega(\log n/\log\log n)$ time. "Bounded memory" means that the memory size is an arbitrary function of n but not of the input values range. It is easy to see that if the memory size is bounded by $O(n^2)$ then the Element Distinctness can be solved in $O(1)$ expected time by using hash functions (seeFact 1 in Section 4). This, however, does not hold for linear size memory. Our parallel hashing algorithm implies that when incorporating randomness, Element Distinctness can be solved in expected $O(\log\log n)$ time using n processors on a collision$^+$-CRCW PRAM (which is weaker than the priority-CRCW model) with linear memory size.

1.3.2 New Design Paradigm.

As a design tool for the hashing algorithms, we present a general paradigm for randomized algorithms which enables one to safely assume that actual behavior in the algorithm run is approximately as expected. It is shown that a randomized algorithm works "as expected", if a prerequisite for an iteration to succeed with a constant probability is that at least a fixed fraction of the previous iterations succeeded. Our result is stronger than the fundamental lemma of Karp, Upfal and Wigderson [24] (which provides a tool for analyzing probabilistic algorithms) in the sense that we allow the performance of each step to be dependent on the performance of the previous steps.

For example, if the size of a problem is known to be halved with constant probability, under the assumption that it has consistently decreased geometrically up to this point, then indeed, a geometric decrease can be assumed for all iterations.

The scheme, called *designing by expectation*, is of independent interest and we expect it to be helpful in the design and analysis of randomized algorithms. It can be useful especially

in fast parallel algorithms, where time constraints may make it impossible to check in run time the actual behavior of certain measures, and one might need to assume that they are approximately as expected. For example, an algorithm that may be less involved by using this scheme is the pattern matching algorithm of Vishkin [29].

1.3.3 Techniques.

Avoiding exact evaluation. The algorithms are based on the 2-level scheme of [14]. Recall that a level-1 function splits the input set into subsets of different sizes. We avoid the exact evaluation of subsets' sizes; such evaluation seems to be inherently slow. Instead, we only make use of the knowledge about the probability distribution of these sizes.

Efficient use of load balancing. The load balancing algorithm of [16] is used to achieve optimal speed-up. A straight forward usage would result in a slowdown of $O(\log\log\log n)$. We demonstrate a more careful usage that results in a slowdown of only $O(\log^* n)$.

1.4 Applications.

Matias and Vishkin [26] proposed using a parallel hashing scheme for space reduction in algorithms in which a large space is required for communication between processors. Such algorithms become space efficient and preserve the same number of operations. The penalties are in becoming randomized and in having some increase in time. Using our hashing scheme, the time increase may be substantially smaller.

There are algorithms for which, by using the scheme of [26], the resulting time increase is $O(\log n)$. By using the new scheme, the time increase is only $O(\log\log n \log^* n)$. This is the case in the construction of *suffix trees* for strings [2, 15] and in the naming assignment procedure for substrings over large alphabets [15].

For other algorithms, the time increase in [26] was $O(\log\log n)$ or $O((\log\log n)^2)$, while the present scheme leaves the expected time unchanged. Such is the case in integer sorting over a polynomial range [23] and over a super-polynomial range [5, 26].

There are applications that could not benefit from $O(\log n)$ time hashing. Such applications are the poly($\log\log n$) time simulations between different models of CRCW PRAM [8, 9]. By using the new hashing scheme, these simulations can be modified to be space efficient and still take poly($\log\log n$) time.

2 Outline of the Algorithms.

Our algorithms construct a 2-level hashing table as in [14]. In the first level, a primary hash function is used to partition S into $O(n)$ *buckets* (subsets) of varying sizes. In the second level, a secondary injective hash function is found for each bucket.

Accordingly, our basic hashing algorithm, **Reduce**, has two main parts. In the first, the keys are grouped into $O(n)$ buckets. The second part of **Reduce** consists of repeated attempts to appropriately map active buckets into allocated memory blocks.

It is shown in [20] that the above process can be done in $O(\log\log n)$ time, while the total space remains $O(n)$. However, it is assumed there that allocation of memory is free. Such allocation depends on the sizes of the buckets. The implementation difficulty is that buckets that remain active after each iteration are determined according to random selections in run time, so a priori memory allocation cannot be performed. On the other hand, an integer sorting algorithm cannot be used (as in [26]) for space allocation since it is too slow for our purpose.

Another assumption in [20] is an $O(1)$ time procedure for counting the number of buckets. The result of this procedure is the basis for the decision on an increase of the block size. The underlining obstacle in a more realistic model implementation is that global coordination is quite limited even in concurrent read/write models. The concurrent read allows broadcast in $O(1)$ time and the concurrent write adds the power of an $O(1)$ time global decision making. However, the bandwidth of those system-wide "communication channels" is quite restricted, and little information can be transmitted in our sub-logarithmic time requirement (see [3]).

We avoid the global cooperation of counting buckets for space allocation by using randomness. As in Thermodynamics, the behavior of a large system of random variables is quite predictable, and this prediction is the source of the global knowledge. A general framework, which we call *designing by expectation*, is provided for planning by the expected behavior.

Memory allocation is done by letting active buckets compete: each trying to get hold of a memory block it picks at random. Algorithm **Reduce** never counts active buckets: the essence of the algorithm is a careful tradeoff between the number of blocks and their size, with respect to the predicted decay of the number of active buckets.

The size of the memory blocks is $O(1)$ in the first iteration, but it grows at each iteration to account for the larger buckets. In early iterations large buckets compete for blocks but cannot succeed in mapping into the blocks. They may succeed only in later iterations, when the memory blocks become appropriately large. The number of "too large" buckets in each iteration is kept small relatively to the number of all competing buckets. To keep this ratio small, the size of memory blocks should be large enough. At each iteration the number of competing buckets is estimated using bounds on the distribution of the buckets' sizes, as generated by a polynomial primary hash function. The number of memory blocks should be appropriately greater than this estimate. There is a clear tradeoff between the number of memory blocks and their sizes.

It is shown that each iteration of Algorithm **Reduce** reduces the total number of active keys by at least a constant factor. We apply $O(\log\log n)$ of those iterations to reduce the number of keys to $O(n/\text{polylog } n)$. Our second algorithm, **WrapUp**, is activated at this point.

Algorithm **WrapUp** can be implemented using the poly-logarithmic ratio between space and the number of keys: a different setting is possible for the parameters of Algorithm **Reduce**

(i.e., for the number and size of memory blocks in each iteration); this setting gives rise to a much faster rate of the decrease in the number of active keys, and by this to a total of $O(\log\log n)$ iterations until all keys become inactive. Another alternative, which we choose to use, is to reallocate the processors to active keys such that $O(\log n)$ processors hash the same key into different memory blocks. This allocation enables a constant time hashing of the remaining active keys. The allocation phase takes $O(\log\log n)$ expected time.

Algorithms **Reduce** and **WrapUp** make different demands on the primary hash function. Hence, the primary hash function they use is different. In order to combine the output of the two algorithms, a special tag is added to each bucket that **Reduce** was not able to resolve. All lookups that end in a tagged bucket restart the search using the hash function constructed by Algorithm **WrapUp**.

Both algorithms, **Reduce** and **WrapUp**, are not optimal in the sense that the time-processor product is greater than $O(n)$. To improve on that we modify Algorithm **Reduce** to achieve an optimal speed-up algorithm for the hashing problem.

The rest of the paper is organized as follows: The design by expectation paradigm is introduced in Section actual-expected. Some necessary definitions and technicalities are in Section 4. Algorithms **Reduce** and **WrapUp** are presented in Sections 5 and 6. The modifications of **Reduce** required for achieving optimal speed-up are described in Section 7. We end with the concluding remarks of Section 8.

3 Designing by Expectation.

Consider an iterative randomized algorithm. After each iteration some natural measure of the problem decreases by a random amount. Karp, Upfal and Wigderson [24] considered the case where a bound on the expectation of the decrease is known, and gave an upper bound on the expected number of iterations that the algorithm requires. Their bound is the best possible in the sense that there are cases for which the upper bound is tight. However, their technique gives global execution estimates and does not provide knowledge about the intermediate performance of the algorithm. Such knowledge may be crucial for the fine design of the next iterations.

We show that in each iteration one can actually assume that in all the previous iterations the algorithm was not too far from its expected behavior. The paradigm suggested is:

> *Design an iteration to be "successful" with a constant probability under the assumption that at least a constant fraction of the previous iterations were "successful".*

LEMMA 1. **(probabilistic induction)** *Let* Alg *be an iterative randomized algorithm, with probability* $\geq 1/2$ *of succeeding at iteration* $(i+1)$*, provided that among the first* i *iterations at least* $i/4$ *were successful. Then, with probability* $\Omega(1)$*, for every* $i > 0$ *the number of successful iterations is at least* $i/4$.

The proof is given in Appendix A.

The notation $\Omega(1)$ for probability is used to emphasize that it is positive *and* constant. (It can be shown that in this case the indefinite success probability is $> 1/3$.) The lemma is not given in its most general or tight form but rather in a way that we feel is convenient to use in an algorithmic design.

The following corollary gives information about the global performance:

COROLLARY 1. *If Algorithm* **Alg** *requires t successful iterations to (successfully) terminate, then its time complexity is $O(t)$ with probability $\Omega(1)$. If it is also possible to detect termination then Algorithm* **Alg** *can be modified to (successfully) terminate in $O(t)$ expected time.*

Typically, an iteration will be considered *successful* if some measure on the problem decreases by a certain amount: Let m_i be the measure after iteration i and g a monotone nondecreasing function such that $g(x) = o(x)$ (e.g., $g(\cdot)$ is $\frac{1}{2}(\cdot), \log(\cdot), \log\log(\cdot)$ etc.). Iteration i is said to *succeed* if $m_{i+1} \leq g(m_i)$. (In our algorithms $E(m_{i+1})$ as a function of m_i is known, under an assumption about m_i, and the probability is deduced from it.) Assume that $m_{i+1} \leq m_i$, $m_0 = m$ and that the algorithm terminates when $m_i \leq 2$. Let $g^{(1)}(x) = g(x)$, $g^{(i)}(x) = g\left(g^{(i-1)}(x)\right)$ for $i > 1$, and $g^*(x) = \min\{i : g^{(i)}(x) \leq 2\}$. For example, if $g(x) = x/2$ then $g^*(x) = \log x$.

After the ith successful iteration, the measure is at most $g^{(i)}(m)$, and the algorithm terminates after $g^*(m)$ iterations.

Under such definitions, the probabilistic induction lemma and corollary can be stated as:

If

$$\forall i > 0 \quad \mathbf{Prob}\left(m_{i+1} \leq g(m_i) \;\middle|\; m_i \leq g^{\lfloor i/4 \rfloor}(m)\right) \geq \frac{1}{2}$$

then

$$\mathbf{Prob}\left(\forall i > 0 \quad m_i \leq g^{\lfloor i/4 \rfloor}(m)\right) = \Omega(1)$$

and the algorithm terminates within at most $4g^*(m)$ iterations with probability $\Omega(1)$.

4 Preliminaries.

4.1 Hash Functions.

We assume that $U = \{0, 1, \ldots, q-1\}$ (recall that U is finite) where q is some prime. Let h be a hash function

$$U \stackrel{h}{\mapsto} [0, \ldots, s-1],$$

then h splits the input set S into buckets $B_i^h := \{x \in S \mid h(x) = i\}$ of sizes $b_i^h = |B_i^h|$, $0 \leq i < s$. We say that h is *c-perfect for* S if $b_i^h \leq c$ for all $0 \leq i < s$; h is called *perfect* for S if it is 1-perfect for it.

We will use the class of d-degree polynomial hash functions

$$H_s^d := \left\{ h \;\middle|\; h(x) := \left(\sum_{i=0}^{d} a_i x^i \bmod q\right) \bmod s,\ a_i \in U \right\}.$$

An easy consequence of the basic lemma of [14] is

FACT 1. *If the input set S is fixed and h is picked at random from the class H_s^1 then*

$$\textbf{Prob}\,(h \text{ is not perfect for } S) \;\leq\; \frac{|S|^2}{s}.$$

We will be using the above for the level-2 functions in our scheme. For the level-1 function we need two classes with somewhat better distribution of buckets' sizes than those of H^1:

FACT 2. ([10]) *Let the input set S be fixed, and h be picked at random from H_s^d, where $d \geq 1$, $s \geq n$. Then*

$$\textbf{Prob}\left(\sum_{i=0}^{s-1} b_i^d \leq c \cdot n\right) \;\geq\; \frac{1}{2},$$

for some constant $c > 0$.

Recently Dietzfelbinger and Meyer auf der Heide [12] showed how polynomial hash functions can be combined to create a new class of hash functions that achieves a distribution of buckets' sizes that is very close to that of truly random functions. The following fact summarizes the aspects of their results which are necessary for our parallel hashing:

FACT 3. ([12]) *There is a class R of hash functions and a subclass $R(S) \subseteq R$ of functions to the range $[0, \ldots, n-1]$ that satisfy the following properties:*

(a) $h \in R$ can be evaluated in constant time.

(b) n processors can pick a random $h \in R$ in $O(1)$ time.

(c) If h is picked at random from R then $\textbf{Prob}\,(h \notin R(S)) = o(1)$.

(d) For $h \in R(S)$, h is $\log n$-perfect with probability $1 - o(1)$.

4.2 Load Balancing.

A problem which turns out to be closely related to parallel hashing is load balancing: Q independent tasks are initially distributed in an unknown manner among P processors. The input to each processor consists of a count of its tasks and a pointer to an array of tasks descriptions. The *load balancing* problem is to redistribute the tasks such that each processor has $O(Q/P)$ tasks. Recently, Gil [17, 16] presented a fast algorithm for load balancing that runs in $O(\log\log\min(P, Q))$ time with high probability.

4.3 Miscellaneous.

Definition of $\log^* n$: Let $\log^{(1)} x = \log x$ and $\log^{(i)} x = \log(\log^{(i-1)} x)$ for $i > 1$, then $\log^* x = \min\{i : \log^{(i)} x \leq 2\}$.

Optimal speed-up: A parallel algorithm is said to have *optimal speed-up* if the product time $\times$ processors (also denoted as *number of operations*) is, up to a constant factor, the same as the time complexity of the best known sequential algorithm for the same problem. We shall say that an algorithm is *optimal* if it has optimal speed-up.

Model of computation: As model of computation we use the concurrent-read concurrent-write parallel random access machine (CRCW PRAM) family. The members of this family

differ by the outcome of the event where more than one processor attempt to write simultaneously into the same shared memory location: in the *common*-CRCW ([25]) all these processors must attempt to write the same value (and this value is written); in the *collision*-CRCW ([13]) a special collision symbol is written in the cell; in the *collision*$^+$-CRCW ([9]) if different values are attempted to be written then a special collision symbol is written in the cell; in the *arbitrary*-CRCW ([28]) one of the processors succeeds, and it is not known in advance which one; in the *priority*-CRCW ([21]) the lowest-numbered processor succeeds.

If all the input elements are distinct, then the collision-CRCW model can be used for all of our algorithms. This model is weaker than the collision$^+$-, the arbitrary- and priority-CRCW machines, and is incomparable with the common-CRCW ([22]).

If the input elements are not distinct (as is typically the case in applications for parallel algorithms) then we use the *collision*$^+$-CRCW model. In fact, the hashing scheme would also work on collision-CRCW but requires an additional pre-processing procedure.

5 Reducing the Number of Active Keys

5.1 Algorithm Reduce.

The input to Algorithm Reduce is a set S of n keys, given in an array. A level-1 function f is selected at random from the class H^{10}_{4n}. f partitions the set S into $4n$ buckets: *bucket* i is the subset of elements that are mapped by f into i. For each bucket, a special memory region (called *block*) is assigned. Then, a level-2 function is found for each bucket. Their descriptions are written in an array *Bucket*.

Each level-2 function should map the elements of its bucket into the assigned block in a one-to-one manner. A bucket is *active* if an appropriate level-2 function has not yet been found, and is inactive otherwise. A key is *active* if its bucket is active.

Initially all keys and buckets are active. The ultimate goal of Algorithm Reduce is to reduce the number of active keys (and hence the number of active buckets) to $O(n/\text{polylog}n)$. The algorithm consists of $O(\log\log n)$ iterations. In each iteration, the number of active buckets is expected to decrease by a constant factor. To achieve such expected decrease, an assumption is being made about the performance of previous iterations. The probabilistic induction lemma (Lemma 1) supplies the basis on which the algorithm is designed and analyzed.

In each iteration i, a new memory region is used. It is partitioned into u_i blocks of size β_i each (The exact values of u_i and β_i will be set later.) The iteration consists of two steps:

Step1 *(reserve)* Each active bucket selects at random one of the u_i blocks. If several buckets select the same block then they all fail. Only buckets that managed to reserve a private block carry on to Step2.

Step2 *(map)* Each bucket randomly selects a function from $H^1_{\beta_i}$, and uses this function for

hashing itself into its reserved block. If the function is injective, then its description is written in the appropriate cell of array *Bucket* and the bucket becomes inactive.

The crux of the algorithm is in giving a proper setting for u_i and β_i. It should be noted that there is a tradeoff between u_i and β_i since the space used in iteration i is $O(u_i\beta_i)$.

5.2 Analysis.

Let m_i be the number of active buckets by the beginning of iteration i ($m_0 = m = 4n$).

LEMMA 2. *For a proper setting of u_i and β_i, Algorithm* **Reduce** *satisfies*

$$\textbf{Prob}\left(\forall i \geq 0 \quad m_i \leq m2^{-i/4}\right) = \Omega(1).$$

Proof The proof is by Lemma 1. Iteration i is *successful* if $m_{i+1} \leq m_i/2$. Thus, the number of active buckets after j successful iterations is $\leq m2^{-j}$. W.l.o.g., we assume that if $m_{i+1} \leq m_i/2$ then $m_{i+1} = m_i/2$ (i.e., if 'too many' buckets become inactive, then some of them are still considered as active). Thus,

$$m_i \geq m2^{-i}. \tag{1}$$

The inductive hypothesis is

$$m_i \leq m2^{-i/4} \tag{2}$$

and the inductive step is to show that

$$\textbf{Prob}\left(m_{i+1} \leq \frac{m_i}{2}\right) \geq \frac{1}{2}. \tag{3}$$

We assume that the level-1 function f satisfies

$$\sum_{j=0}^{s-1} b_j^{10} \leq 2^a \cdot m \tag{4}$$

for some constant $a > 0$. By Fact 2, eq. (4) holds with probability $\geq 1/2$. We set

$$u_i = m2^{4-i/4} \qquad \beta_i = 2^{4+a/5+i/5} \tag{5}$$

In each iteration we track buckets of size at most

$$s_i = \sqrt{\beta_i/8} = 2^{\frac{1}{2}(1+a/5+i/5)}. \tag{6}$$

1. Let m_i' be the number of buckets that are larger than s_i. By eq. (4), $m_i' \cdot s_i^{10} \leq \sum_{j=0}^{s-1} b_j^{10} \leq 2^a m$. Therefore, by eq. (6),

$$m_i' \leq \frac{2^a m}{s_i^{10}} = m2^{a-\frac{10}{2}(1+a/5+i/5)} = m2^{-5-i}. \tag{7}$$

Let $p_0(i)$ be the probability that a bucket will not be of proper size $\leq s_i$. By ineq. (1) and (7),

$$p_0(i) = \frac{m_i'}{m_i} \leq \frac{m2^{-5-i}}{m2^{-i}} = 2^{-5} < \frac{1}{16}.$$

2. Let $p_1(i)$ be the probability that a bucket does not successfully reserve a space block in Step1. By ineq. (2) and eq. (5)

$$p_1(i) \leq \frac{m_i}{u_i} \leq \frac{m2^{-i/4}}{m2^{4-i/4}} = 2^{-4} = \frac{1}{16}.$$

3. Let $p_2(i)$ be the probability of a bucket of size smaller than s_i to be successfully mapped into a block of size β_i in Step2. By Fact 1 and eq. (6)

$$p_2(i) \leq \frac{s_i^2}{\beta_i} = \frac{1}{8}.$$

A bucket of size smaller than s_i that successfully reserves a space block of size β_i, and that is successfully mapped into it, becomes inactive. Therefore, the probability for an arbitrary active bucket at iteration i to remain active at iteration $i+1$ is bounded by

$$p_0(i) + p_1(i) + p_2(i) \leq \frac{1}{8} + \frac{1}{16} + \frac{1}{16} = \frac{1}{4}.$$

The expected number of active buckets in iteration $i+1$ is therefore

$$\mathbf{E}(m_{i+1}) \leq \frac{m_i}{4} \ .$$

By Markov inequality

$$\mathbf{Prob}\left(m_{i+1} \leq \frac{m_i}{2}\right) \geq \frac{1}{2} \ .$$

□

Let n_i be the number of active keys by the beginning of iteration i ($n_0 = n$). We have

COROLLARY 2. *Algorithm* Reduce *satisfies*

$$\mathbf{Prob}\left(\forall i \geq 0 \quad n_i \leq cn2^{-\alpha i}\right) = \Omega(1)$$

for some constants $c, \alpha > 0$.

Proof If follows from eq. (4) (by using Jensen's inequality) that n_i is maximal when all buckets are of the same size x_i. In this case, by eq. (4),

$$m_i \cdot x_i^{10} \leq 2^a \cdot m$$

and

$$n_i = m_i \cdot x_i \leq m_i \cdot \left(\frac{2^a m}{m_i}\right)^{0.1} \leq (2^a m)^{0.1} m_i^{0.9}.$$

Therefore, by Lemma 2, the corollary follows. □

By Corollary 2 we have a geometric decrease in the number of keys with probability $\Omega(1)$. Therefore, after $O(\log\log n)$ expected number of iterations the number of active keys becomes $n/\text{polylog}n$ as required.

Memory usage. The space used in Algorithm Reduce is $\sum_i u_i\beta_i = m2^{8+a/5}\sum_i 2^{i/5-i/4} = O(n)$.

6 Final Step.

After the execution of Algorithm Reduce, the available resources (memory cells and processors) stand in poly-logarithmic ratio to the number of active keys. This resources redundancy permits several $O(\log\log n)$ time implementations of Algorithm WrapUp which takes care of the hashing of the remaining keys. For example, it is possible to use the comparatively

large available memory; the u_i and β_i parameters of Algorithm Reduce can be so set that the fraction of active keys is raised to a constant power in each iteration, which leads to a doubly-logarithmic rate of decrease in the number of keys.

We give a cleaner implementation of Algorithm WrapUp by using the extra computing power:

LEMMA 3. *Suppose there is a team of* $\log n$ *numbered processors allocated to each key, and further suppose that* $2\log^3 n$ *space is allocated to each bucket. Then hashing can be done in constant expected time.*

Proof Use a function from the class R to create n buckets of a size smaller than $\log n$ each. Each bucket uses $\log n$ memory blocks of size $2\log^2 n$ each. For each bucket, $\log n$ hashing attempts are done simultaneously to its blocks. Those attempts are carried by the $\log n$ processors assigned to each key. If none of the attempts succeeds then the bucket fails, otherwise one of the succeeding attempts is picked as the secondary hash function for the bucket (by using the $O(1)$ integer maximum algorithm [13]).

The failure probability of a single attempt is $\leq 1/2$, and the probability that a certain bucket will fail in all its attempts is $O(1/n)$. The expected number of failing buckets is $O(1)$, therefore with $\Omega(1)$ probability no bucket fails, and hence the expected run time is also $O(1)$. □

To complete the description of Algorithm WrapUp it must be shown that the allocation of teams of processors to keys and memory blocks to buckets can be done in $O(\log\log n)$ time.

Allocation can be done by using the $O(\log\log n)$ load balancing algorithm. If each active key is considered as $\log n$ tasks then by distributing these tasks among the processors, each active key will be allocated with $\log n$ processors. Memory allocation can be done by using similar ideas. An even simpler allocation can be achieved by mapping the active keys into an array of size $O(n/\log n)$ in a one to one manner. Such mapping algorithm is, in fact, a building block in the load balancing algorithm, and it can be done in expected $O(\log\log n)$ time. Memory allocation can be done in a similar manner.

7 Achieving Optimal Speed-up.

In order to reach optimal speed-up, the keys array and the buckets array are divided into P sectors, each having an allocated processor. Each iteration in Algorithm Reduce is executed in $O(1)$ sector traversals of each processor.

During the algorithm the number of keys and the number of buckets drop, and accordingly many of the sectors shrink. However, dependency between sectors implies that an iteration's time is proportional to the size of the largest sector. To overcome this obstacle load balancing is used.

Assume that we have a load balancing algorithm that, using P processors, takes $O(x)$ expected time, where $x \geq \log\log n$. While incorporating load balancing into Algorithm **Reduce**, the exact values of the number of keys and the number of buckets are not known in each iteration. Instead we assume the bounds obtained in Lemma 2 and Corollary 2. As noted before this assumption is safe since it occurs with $\Omega(1)$ probability. In the sequel we deal with sectors of keys only. Load balancing of the buckets arrays can easily be inferred.

If load balancing is applied every $O(1)$ iterations, then optimal speed-up can be obtained for $P \leq n/x\log x$. To achieve optimal speed-up for values of P which are as large as $O(n/x\log^* x)$, note that it is "justified" to use an $O(x)$ time load balancing only after the "host" algorithm executed for $\Theta(x)$ time. Specifically, if the sectors are of size y, then the next load balancing procedure should be employed only after $\lceil x/y \rceil$ iterations since in each iteration, traversal takes $O(y)$. We modify the algorithm accordingly, and show that this approach yields an $O(x\log^* x)$ expected time algorithm with optimal speed-up.

Assume that we have $P = n/x\log^* x$ processors, each responsible for a key-sector of size $O(x\log^* x)$. First, load balancing is used after each iteration until all sectors are of size $O(x)$. This will take expected $O(\log\log^* x)$ iterations, based on Corollary 2. After that the algorithm runs in phases; phase i starts with ($O(x)$ time) load balancing, followed by t_i iterations. Let x_i be a bound on the sectors size in phase i ($x_0 = x$). Each traversal takes $O(x_i)$ time. t_i is set to satisfy $x_i t_i = x$, so the total execution time of a phase is $O(x)$.

To bound the number of phases, Corollary 2 is used to obtain

$$x_{i+1} = \frac{x_i}{2^{O(t_i)}} \qquad \text{and hence} \qquad t_{i+1} = t_i 2^{O(t_i)}.$$

For some $i = O(\log^* x)$, $t_i = x$ and $x_i = 1$. The above procedure takes $O(x\log^* x)$ expected total time.

We showed that using a load balancing algorithm with $O(x)$ expected running time, our optimal hashing scheme takes $O(x\log^* x)$ expected time. Using the load balancing algorithm of [16], where $x = \log\log n$, we get an optimal speed-up algorithm that takes $O(\log\log n\log^* n)$ expected time and $O(n)$ expected number of operations.

8 Conclusions.

We presented an algorithm that constructs a perfect hash function for n elements in $O(\log\log n)$ time, using n processors. Our algorithm circumvents the $\Omega(\log n/\log\log n)$ lower bound for counting and sorting by using probabilistic estimations instead of exact counting.

A general paradigm for the design of randomized algorithms was introduced. The paradigm, called *designing by expectation*, is used in the design of our algorithm and may be useful in the design of other fast randomized parallel algorithms.

By using a fast load balancing algorithm we modified the hashing algorithm and obtained optimal speed-up. We demonstrated a careful usage of the load balancing algorithm. This leads to only a small increase in time for the optimal hashing algorithm.

Techniques that are similar to the techniques that are used in this paper can be used to simulate an arbitrary-CRCW on a collision-CRCW. The simulation algorithm takes $O(\log\log n)$ time and $O(n)$ space [18]. Consequently, all the duplications in the input can be eliminated. As a result, the hashing algorithm works also on collision-CRCW.

Further research. The problem considered in this paper is the *static* hashing problem. The *dynamic hashing* (*dictionary*) problem is to maintain a data structure that supports the instructions *insert*, *delete* and *lookup*. In other words, the input set S is changing dynamically. The fastest optimal dictionary is due to [11] and its time complexity is $O(n^{\epsilon})$ for any constant $\epsilon > 0$. The static hashing scheme presented in this paper can be extended to a dynamic hashing scheme with similar complexities [19].

Postscript. Recently [27] presented a new paradigm for randomized parallel algorithms that requires expected $O(\log^* n)$ rounds. This alternative approach has implications to parallel hashing.

Acknowledgments. We thank Aviad Cohen and Jeanette P. Schmidt for helpful comments on a previous version of the paper. We also wish to thank our advisors, Uzi Vishkin and Avi Wigderson, for fruitful discussions and for their encouragement.

References

[1] D. Angluin and L. G. Valiant. Fast probabilistic algorithms for hamiltonian paths and matchings. *J. Comp. Syst. Sci.*, 18:155–193, 1979.

[2] A. Apostolico, C. Iliopoulos, G. M. Landau, B. Schieber, and U. Vishkin. Parallel construction of a suffix tree. *Algorithmica*, 3:347–365, 1988.

[3] P. Beame. Limits on the power of concurrent-write parallel machines. In *Proc. of the 18th Ann. ACM Symp. on Theory of Computing*, pages 169–176, 1986.

[4] P. Beame and J. Hastad. Optimal bounds for decision problems on the CRCW PRAM. In *Proc. of the 19th Ann. ACM Symp. on Theory of Computing*, pages 83–93, 1987.

[5] P. C. P. Bhatt, K. Diks, T. Hagerup, V. C. Prasad, T. Radzik, and S. Saxena. Improved deterministic parallel integer sorting. Technical Report TR 15/1989, Fachbereich Informatik, Universität des Saarlandes, D-6600 Saarbrücken, W. Germany, November 1989.

[6] R. B. Boppana. Optimal separations between concurrent-write parallel machines. In *Proc. of the 21st Ann. ACM Symp. on Theory of Computing*, pages 320–326, 1989.

[7] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Math. Statistics*, 23:493–507, 1952.

[8] B. S. Chlebus, K. Diks, T. Hagerup, and T. Radzik. Efficient simulations between concurrent-read concurrent-write PRAM models. In *Mathematical Foundations of Computer Science*, 1988.

[9] B. S. Chlebus, K. Diks, T. Hagerup, and T. Radzik. New simulations between CRCW PRAMs. In *Fundamentals of Computation Theory, Intl' Conf*, August 1989.

[10] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: upper and lower bounds. In *Proc. of the 29th IEEE Annual Symp. on Foundation of Computer Science*, pages 524–531, October 1988. Also, Revised Version: Tech. Report, University of Paderborn, FB 17 Mathematik/Informatik, 1991.

[11] M. Dietzfelbinger and F. Meyer auf der Heide. An optimal parallel dictionary. In *Proc. of the 1989 Symposium on Parallel Algorithms and Architectures*, pages 360–368, 1989.

[12] M. Dietzfelbinger and F. Meyer auf der Heide. A new universal class of hash functions and dynamic hahshing in real time. In *Proc. of 17th ICALP, Springer LNCS 443*, pages 6–19, 1990.

[13] F. E. Fich, P. L. Ragde, and A. Wigderson. Relations between concurrent-write models of parallel computation. *SIAM J. Comput.*, 1:606–627, 1988.

[14] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of The Association for Computing Machinery*, 31:538–544, July 1984.

[15] Z. Galil and R. Giancarlo. Data structures and algorithms for approximate string matching. *J. of Complexity*, 4:33–72, 1988.

[16] J. Gil. Fast load balancing on pram. manuscript, 1990.

[17] J. Gil. *Lower Bounds and Algorithms for Hashing and Parallel Processing.* PhD thesis, The Hebrew University of Jerusalem, Givat Ram 91904, Jerusalem, Israel, November 90.

[18] J. Gil and Y. Matias. Fast and efficient simulations among CRCW models. In preparation, 1990.

[19] J. Gil, Y. Matias, and U. Vishkin. A fast parallel dictionary. In preparation, 1990.

[20] J. Gil, F. Meyer auf der Heide, and A. Wigderson. Not all keys can be hashed in constant time. In *Proc. of the 22st Ann. ACM Symp. on Theory of Computing*, pages 244–253, 1990.

[21] L. M. Goldschlager. A universal interconnection pattern for parallel computers. *Journal of The Association for Computing Machinery*, 29:1073–1086, 1982.

[22] V. Grolmusz and P. L. Ragde. Incomparability in parallel computation. In *Proc. of the 28th IEEE Annual Symp. on Foundation of Computer Science*, pages 89–98, 1987.

[23] T. Hagerup. Towards optimal parallel bucket sorting. *Information and Computation*, 75:39–51, 1987.

[24] R. M. Karp, E. Upfal, and A. Wigderson. The complexity of parallel search. *J. Computer and System Sciences*, 36(2):225–253, April 1988.

[25] L. Kučera. Parallel computation and conflicts in memory access. *Information Processing Letters*, 14:93–96, 1982.

[26] Y. Matias and U. Vishkin. On parallel hashing and integer sorting. Technical Report TR-158/89, Eskenasy Inst. of Computer Sciences, Tel-Aviv Univ., December 1989. Also in UMIACS-TR-90-13, Inst. for Advanced Computer Studies, Univ. of Maryland, Jan. 1990.

[27] Y. Matias and U. Vishkin. Converting high probability into nearly-constant time – with applications to parallel hashing. Extended Abstract, November, 1990.

[28] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *J. Algorithms*, 3:57–67, 1982.

[29] U. Vishkin. Deterministic sampling - a new technique for fast pattern matching. In *Proc. of the 22st Ann. ACM Symp. on Theory of Computing*, 1990. Also to appear in SIAM J. Computing.

A Proof of Probabilistic Induction Lemma.

Lemma 1 is proved by examining an infinite sequence of indicator random variables that represent the success/failure of the algorithm's iterations.

DEFINITION 1. *Let $X = \langle x_1, x_2, \ldots \rangle$ be an infinite sequence of indicator random variables, and $X_i = \langle x_1, \ldots, x_i \rangle$ be the ith prefix of X.*

- *X_u is* balanced *if $\sum_{i=1}^{u} x_i \geq u/4$, and otherwise it is* unbalanced.
- *X_u is* strongly balanced *if X_i is balanced for every $i \leq u$.*
- *X is* always balanced *if, for all $u > 0$, X_u is balanced.*

We will use the well known Chernoff bounds:

LEMMA 4. ([7, 1]) *Let Y be a binomial. Then*

$$\mathbf{Prob}\left(Y < \frac{E(Y)}{2}\right) \leq e^{-E(Y)/8}. \tag{8}$$

LEMMA 5. *If $\mathbf{Prob}(x_i = 1) \geq 1/2$, for $i \geq 1$, and x_i are independent then X is always balanced with probability $\Omega(1)$.*

Proof Let v be fixed. The event that X is always balanced is the intersection of two events: X_v is strongly balanced and X_u is never unbalanced for $u \geq v$. Although these events are dependent, their dependency is in the desired direction. The first event occurs with probability $\Omega(1)$ since v is fixed. It remains to show the existence of such (fixed) v for which the second event occurs with probability $\Omega(1)$ as well.

The sum $\sum_{i=1}^{u} x_i$ is bounded by a binomial with parameters $(u, 1/2)$. Therefore, by Chernoff ineq. (8),

$$\begin{aligned} \mathbf{Prob}(X_u \text{ is unbalanced}) &= \mathbf{Prob}\left(\sum_{i=1}^{u} x_i < u/4\right) \\ &\leq e^{-u/16} . \end{aligned}$$

Consequently, $p(v)$, the probability that there exists $u \geq v$ for which X_u is unbalanced, is bounded by

$$p(v) \leq \sum_{u=v}^{\infty} e^{-u/16} = \frac{e^{-v/16}}{(1 - e^{-1/16})} .$$

Clearly there is a fixed v for which $1 - p(v) = \Omega(1)$. (The existence of such fixed v can be also obtained from the strong law of large numbers.) □

We are interested in the case that $\mathbf{Prob}(x_i = 1) \geq 1/2$ is true only under the condition that X_{i-1} is strongly balanced. We will analyze the more restrictive case that $\mathbf{Prob}(x_i = 1 | X_i \text{ is strongly balanced}) = \frac{1}{2}$.

Let $SB(i)$ be the set of all assignments to the vector X_i for which X_i is strongly balanced. Note that

$$\begin{aligned}\textbf{Prob}\,(X_u \text{ is strongly balanced}) &= \textbf{Prob}\,(X_u \in SB(u)) \\ &= \sum_{e \in SB(u)} \textbf{Prob}\,(X_u = e)\ . \end{aligned} \tag{9}$$

DEFINITION 2. *We say that X is* partially independent *if*

$$\textbf{Prob}\,(x_1 = 1) = \textbf{Prob}\,(x_1 = 0) = 1/2 \tag{10}$$

and

$$\forall e \in SB(i)\ \textbf{Prob}\,(x_{i+1} = 1 \mid X_i = e) = 1/2\ . \tag{11}$$

LEMMA 6. *If X is partially independent then X is always balanced with probability $\Omega(1)$.*

Proof Let $Y = \langle y_1, y_2, \ldots \rangle$ be an infinite sequence of indicator random variables such that y_i are independent and $\textbf{Prob}\,(()y_i = 1) = 1/2$, for $i \geq 1$. Intuitively, X behaves as well as Y, as long as X is strongly balanced. More formally, the proof will show that

$$\forall u \geq 1,\ \textbf{Prob}\,(X_u \in SB(u)) = \textbf{Prob}\,(Y_u \in SB(u))\ .$$

Using (9) it is enough to show that for every $e \in SB(u)$

$$\textbf{Prob}\,(X_u = e)\ =\ \textbf{Prob}\,(Y_u = e) \tag{12}$$

which is proved by induction on u. In the case $u = 1$ the set $SB(u)$ is $\{\langle 1 \rangle\}$. By (10)

$$\textbf{Prob}\,(x_1 = 1)\ =\ \textbf{Prob}\,(y_1 = 1)\ =\ 1/2.$$

Assuming by induction that (12) holds for $u = i$, we show that it holds for $u = i+1$. Let $e = \langle e_1, \ldots, e_{i+1} \rangle \in SB(i+1)$ and let $e' = \langle e_1, \ldots, e_i \rangle$. Then, by definition, $e' \in SB(i)$. Using (11), the inductive hypothesis, and the fact that y_{i+1} is an independent random variable we get that

$$\begin{aligned}\textbf{Prob}\,(X_{i+1} = e) &= \textbf{Prob}\,(X_i = e' \cap x_{i+1} = e_{i+1}) \\ &= \frac{1}{2}\textbf{Prob}\,(X_i = e') \\ &= \frac{1}{2}\textbf{Prob}\,(Y_i = e') \\ &= \textbf{Prob}\,(Y_i = e' \cap y_{i+1} = e_{i+1}) \\ &= \textbf{Prob}\,(Y_{i+1} = e)\end{aligned}$$

□

Usually X will satisfy a stronger condition, and a weaker statement will therefore suffice:

COROLLARY 3. *If $\textbf{Prob}\,(x_u = 1) \geq 1/2$ when X_{u-1} is balanced, then X is always balanced with probability $\Omega(1)$.*

Lemma 1 now follows.