Simple Fast Parallel Hashing by Oblivious Execution *

Joseph Gil †
Dept. of Computer Science
The Technion, Israel
Technion City, Haifa 32000
ISRAEL

Yossi Matias ‡
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974
USA

September 6, 1995

Abstract

A *hash table* is a representation of a set in a linear size data structure that supports constant-time membership queries. We show how to construct a hash table for any given set of n keys in $O(\lg\lg n)$ parallel time with high probability, using n processors on a weak version of a CRCW PRAM. Our algorithm uses a novel approach of hashing by "oblivious execution" based on probabilistic analysis to circumvent the parity lower bound barrier at the near-logarithmic time level. The algorithm is simple and is sketched by the following:

1. Partition the input set into buckets by a random polynomial of constant degree.
2. For $t := 1$ to $O(\lg\lg n)$ do
   (a) Allocate M_t memory blocks, each of size K_t.
   (b) Let each bucket select a block at random, and try to injectively map its keys into the block using a random linear function. Buckets that fail carry on to the next iteration.

The crux of the algorithm is a careful a priori selection of the parameters M_t and K_t. The algorithm uses only $O(\lg\lg n)$ random words, and can be implemented in a work-efficient manner.

*Parts of this research were presented in preliminary forms in the *Second Annual ACM-SIAM Symposium on Discrete Algorithms*, January 1991 [21], and in the *21st International Colloquium on Automata, Languages and Programming*, July 1994 [25].

†Part of research was done while author was at the Hebrew University and at the University of British Columbia. E-mail address: `yogi@cs.technion.ac.il`.

‡Part of research was done while author was at Tel Aviv University and at the University of Maryland Institute for Advanced Computer Studies, and was partially supported by NSF grants CCR-9111348 and CCR-8906949. E-mail address: `matias@research.att.com`.

1 Introduction

Let S be a set of n keys drawn from a finite universe U. The *hashing* problem is to construct a function $H : U \rightarrow [0, m-1]$ with the following attributes:

Injectiveness: no two keys in S are mapped by H to the same value,

Space efficiency: both m and the space required to represent H are $O(n)$, and

Time efficiency: for every $x \in U$, $H(x)$ can be evaluated in $O(1)$ time by a single processor.

Such a function induces a linear space data structure, a *perfect hash table*, for representing S. This data structure supports membership queries in $O(1)$ time.

This paper presents a simple, fast and efficient parallel algorithm for the hashing problem. Using n processors, the running time of the algorithm is $O(\lg\lg n)$ with overwhelming probability, and it is superior to previously known algorithms in several respects.

Computational models As a model of computation we use the concurrent-read concurrent-write parallel random access machine (CRCW PRAM) family (see, e.g., [35]). The members of this family differ by the outcome of the event where more than one processor attempts to write simultaneously into the same shared memory location. The main sub-models of CRCW PRAM in descending order of power are: the PRIORITY ([29]) in which the lowest-numbered processor succeeds; the ARBITRARY ([42]) in which one of the processors succeeds, and it is not known in advance which one; the COLLISION$^+$ ([9]) in which if different values are attempted to be written, a special collision symbol is written in the cell; the COLLISION ([15]) in which a special collision symbol is written in the cell; the TOLERANT ([32]) in which the contents of that cell do not change; and finally, the less standard ROBUST ([7, 34]) in which if two or more processors attempt to write into the same cell in a given step, then, after this attempt, the cell can obtain any value.

1.1 Previous Work

Hash tables are fundamental data structures with numerous applications in computer science. They were extensively studied in the literature; see, e.g., [37, 40] for a survey or [41] for a more recent one. Of particular interest are perfect hash tables, in which every membership query is guaranteed to be completed in constant time in the worst case. Perfect hash tables are perhaps even more significant in the parallel context, since the time for executing a batch of queries in parallel is determined by the slowest query.

Fredman, Komlós, and Szemerédi [16] were the first to solve the hashing problem in expected linear time for any universe size and any input set. Their scheme builds a *2-level* hash function: a *level-1* function splits S into subsets ("buckets") whose sizes are distributed in a favorable manner. Then, an injective *level-2* hash function is built for each subset by allocating a private memory block of an appropriate size.

This 2-level scheme formed a basis for algorithms for a dynamic version of the hashing problem, also called the *dictionary problem*, in which insertions and deletions may change S dynamically. Such algorithms were given by Dietzfelbinger, Karlin, Mehlhorn, Meyer auf der Heide, Rohnert and Tarjan [12], Dietzfelbinger and Meyer auf der Heide [14], and by Dietzfelbinger, Gil, Matias and Pippenger [11].

In the parallel setting, Dietzfelbinger and Meyer auf der Heide [13] presented an algorithm for the dictionary problem. for each fixed $\epsilon \geq 0$, n arbitrary dictionary instructions (insert, delete, or lookup), can be executed in $O(n^{\epsilon})$ expected time on a a $n^{1-\epsilon}$-processor PRIORITY CRCW. Matias and Vishkin [39] presented an algorithm for the hashing problem that runs in $O(\lg n)$ expected time using $O(n/\lg n)$ processors on an ARBITRARY CRCW. This was the fastest parallel hashing algorithm previous to our work. It is based on the 2-level scheme and makes extensive use of counting and sorting procedures.

The only known lower bounds for parallel hashing were given by Gil, Meyer auf der Heide and Wigderson [27]. In their (rather general) model of computation, the required number of parallel steps is $\Omega(\lg^* n)$. They also showed that in a more restricted model, where at most one processor may simultaneously work on a key, parallel hashing time is $\Omega(\lg\lg n)$. They also gave an algorithm which yields a matching upper bound if only function applications are charged and all other operations (e.g., counting and sorting) are free. Our algorithm falls within the realm of the above mentioned restricted model and matches the $\Omega(\lg\lg n)$ lower bound while charging for all operations on the concrete PRAM model.

1.2 Results

Our main result is that a linear static hash table can be constructed in $O(\lg\lg n)$ time with high probability and $O(n)$ space, using n processors on a CRCW PRAM. Our algorithm has the following properties:

Time optimality It is the best possible result that does not use processor reallocations, as shown in [27]. Optimal speed-up can be achieved with a small penalty in execution time. It is a significant improvement over the $O(\lg n)$ time algorithm of [39].

Reliability Time bound $O(\lg\lg n)$ is obeyed with high probability; in contrast, the time bound of the algorithm in [39] is guaranteed only with constant probability.

Simplicity It is arguably simpler than any other hashing algorithm previously published. (Nevertheless, the analysis is quite involved due to tight tradeoffs between the probabilities of conflicting events.)

Reduced randomness It is adapted to consume only $O(\lg\lg n)$ random words, compared to $\Omega(n)$ random words that were previously used.

Work optimality A work optimal implementation is presented, in which the time-processor product is $O(n)$ and the running time is increased by a factor of $O(\lg^* n)$; it also requires only $O(\lg\lg n)$ random words.

Computational model If we allow lookup time to be $O(\lg\lg n)$ as well, then our algorithm can be implemented on the ROBUST CRCW model.

Our results can be summarized in the following theorem.

Theorem 1 *Given a set of n keys drawn from a universe U, the hashing problem can be solved using $O(n)$ space: (i) in $O(\lg\lg n)$ time with high probability, using n processors, or (ii) in $O(\lg\lg n \lg^* n)$ time and $O(n)$ operations with high probability. The algorithms run on a* CRCW PRAM *where no reallocation of processors to keys is employed, and use $O(\lg\lg |U| + \lg n \lg\lg n)$ random bits.*

The previous algorithms implementing the 2-level scheme, either sequentially or in parallel, are based on grouping the keys according to the buckets to which they belong, and require learning the size of each bucket. Each bucket is then allocated a private memory block whose size is dependent on the bucket size. This approach relies on techniques related to sorting and counting, which require $\Omega(\lg n/\lg\lg n)$ time to be solved by polynomial number of processors, as implied by the lower bound of Beame and Hastad [4]. This lower bound holds even for randomized algorithms. (More recent results have found other, more involved, ways to circumvent these barriers; cf. [38, 3, 26, 30].)

We circumvent the obstacle of learning buckets sizes for the purpose of appropriate memory allocation by a technique of *oblivious execution*, sketched by Figure 1.

1. Partition the input set into buckets by a random polynomial of constant degree.
2. For $t := 1$ to $O(\lg\lg n)$ do
    (a) Allocate M_t memory blocks, each of size K_t.
    (b) Let each bucket select a block at random, and try to injectively map its keys into the block using a random linear function; if the same block was selected by another bucket, or if no injective mapping was found, then the bucket carries on to the next iteration.

Figure 1: *The template for the hashing algorithm.*

The crux of the algorithm is a careful a priori selection of the parameters M_t and K_t. For each iteration t, M_t and K_t depend on the expected number of active buckets and the expected distribution of bucket sizes at iteration t in a way that makes the desired progress possible (or rather, likely).

The execution is oblivious in the following sense: All buckets are treated equally, regardless of their sizes. The algorithm does not make any explicit attempt to estimate the sizes of individual buckets and to allocate memory to buckets based on their sizes, as is the case in the previous implementations of the 2-level scheme. Nor does it attempt to estimate the number of active buckets or the distribution of their sizes.

The selection of the parameters M_t and K_t in iteration t is made according to a priori estimates of the above random variables. These estimates are based on properties of the level-1 hash function as well as on inductive assumptions about the behavior of previous iterations.

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 machine with bounded memory requires $\Omega(\lg n / \lg\lg n)$ time. "Bounded memory" means that the memory size is an arbitrary function of n but not of the range of the input values. It is easy to see that if the memory size is bounded by $\Omega(n^2)$ then Element Distinctness can be solved in $O(1)$ expected time by using hash functions (Fact 2.2). 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 $O(\lg\lg n)$ time using n processors on COLLISION$^+$ (which is weaker than the PRIORITY model) with linear memory size.

1.3 Applications

The perfect hash table data structure is a useful tool for parallel algorithms. Matias and Vishkin [39] proposed using a parallel hashing scheme for space reduction in algorithms in which a large amount of space is required for communication between processors. Such algorithms become space efficient and preserve the number of operations. The penalties are in introducing randomization 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 [39], the resulting time increase is $O(\lg n)$. By using the new scheme, the time increase is only $O(\lg\lg n \lg^* n)$. This is the case in the construction of *suffix trees* for strings [2, 17] and in the naming assignment procedure for substrings over large alphabets [17].

For other algorithms, the time increase in [39] was $O(\lg\lg n)$ or $O((\lg\lg n)^2)$, while our algorithm leaves the expected time unchanged. Such is the case in integer sorting over a polynomial range [33] and over a super-polynomial range [5, 39].

More applications are discussed in the conclusion section.

1.4 Outline

The rest of the paper is organized as follows. Preliminary technicalities used in our algorithm and its analysis are given in Section 2. The algorithm template is presented in greater detail in Section 3. Two different implementations, based on different selections of M_t and K_t, are given in the subsequent sections. Section 4 presents an implementation that does not fully satisfy the statements of Theorem 1 but has a relatively simple analysis. An improved implementation of the main algorithm, with more involved analysis, is presented in Section 5. In Section 6 we show how to reduce the number of random bits. Section 7 explains how the algorithm can be implemented with an optimal number of operations. The model of computation is discussed in Section 8, where we also give a modified algorithm for a weaker model. Section 9 briefly discusses the extension of the hashing problem, in which the input may consist of a multi-set. Finally, conclusions are given in Section 10.

2 Preliminaries

The following inequalities are standard (see, e.g. [1]):

Markov's inequality Let ω be a random variable assuming non-negative values only. Then

$$\mathbf{Prob}\,(\omega > T) < \mathbf{E}\,(\omega)/T \ . \tag{1}$$

Chebyshev's inequality Let ω be a random variable. Then, for $T > 0$,

$$\mathbf{Prob}\,(|\omega - \mathbf{E}\,(\omega)| > T) < \mathbf{Var}\,(\omega)/T^2 \ . \tag{2}$$

Chernoff's inequality Let ω be a binomial variable. Then, for $T > 0$,

$$\mathbf{Prob}\,(|\omega - \mathbf{E}\,(\omega)| > T) = e^{-\Omega(T^2/\mathbf{E}(\omega))} \ . \tag{3}$$

Terminology for probabilities We say that an event occurs with *n-dominant probability* if it occurs with probability $1 - n^{-\Omega(1)}$. Our usage of this notation is essentially as follows. If a poly-logarithmic number of events are such that each one of them occurs with n-dominant probability, then their conjunction occurs with n-dominant probability as well. We will therefore usually be satisfied by demonstrating that each algorithmic step succeeds with n-dominant probability.

Fact 2.1 *Let $\omega_1, \ldots, \omega_n$ be pairwise independent binary random variables, and let $\omega = \sum_{1 \le i \le n} \omega_i$. Let $\epsilon > 0$ be constant; let $0 < \xi_1 < (1-\epsilon)\mathbf{E}\,(\omega)$ and $\xi_2 > (1+\epsilon)\mathbf{E}\,(\omega)$. Then*

1. *$\omega > \xi_1$ with ξ_1-dominant probability,*

2. *$\omega < \xi_2$ with ξ_2-dominant probability, and*

3. *$(1-\epsilon)\mathbf{E}(\omega) \le \omega \le (1+\epsilon)\mathbf{E}(\omega)$ with $\mathbf{E}(\omega)$-dominant probability.*

Proof. Recall the well known fact that

$$\begin{aligned}
\mathbf{Var}(\omega) &= \sum_{1\le i\le n} \mathbf{Var}(\omega_i) && \text{since } \omega_i \text{ are pairwise independent} \\
&\le \sum \mathbf{E}(\omega_i) && \text{since } 0 \le \omega_i \le 1 \\
&= \mathbf{E}(\omega) \ .
\end{aligned} \tag{4}$$

1. By Inequality (2)

$$\mathbf{Prob}(\omega > \xi_1) \le \mathbf{Prob}(|\omega - \mathbf{E}(\omega)| > \epsilon\mathbf{E}(\omega)) < 1/\epsilon^2\mathbf{E}(\omega) < 1/\epsilon^2\xi_1 \ .$$

2. If $\xi_2 \le \mathbf{E}(\omega)^2$ then by Inequality (2)

$$\mathbf{Prob}(\omega > \xi_2) \le \mathbf{Prob}(|\omega - \mathbf{E}(\omega)| > \epsilon\mathbf{E}(\omega)) < 1/\epsilon^2\mathbf{E}(\omega) < 1/\epsilon^2\sqrt{\xi_2} \ .$$

   If $\xi_2 > \mathbf{E}(\omega)^2$ then by Inequality (1)

$$\mathbf{Prob}(\omega > \xi_2) < \mathbf{E}(\omega)/\xi_2 < 1/\sqrt{\xi_2} \ .$$

3. Follows immediately from the above.

∎

Hash functions For the remainder of this section, let $S \subseteq U$ be fixed, $|S| = n$. A hash function $h : U \to [0, m-1]$ splits the set S into buckets; *bucket* i is the subset $\{x \in S \mid h(x) = i\}$ and its size is $s_i = \left|S \cap h^{-1}(i)\right|$, for $0 \le i < m$. An element $x \in S$ *collides* if its bucket is not a singleton. The function is *injective*, or *perfect*, if no element collides. Let

$$B_r = B_r(h) = \sum_{0\le i<m} \binom{s_i}{r} \ . \tag{5}$$

A function is injective if and only if $B_2 = 0$, since B_2 is the number of collisions of pairs of keys. More generally, B_r is the number of r-tuples of keys that collide under h.

Polynomial hash functions Let $U = [0, u-1]$ where u is prime. The class of degree-d polynomial hash functions, $d \ge 1$, mapping U into $[0, m-1]$ is

$$\mathcal{H}_m^d := \left\{ h \;\middle|\; h(x) = \left(\sum_{i=0}^{d} c_i x^i \bmod u\right) \bmod m, \text{ for some } c_0, \ldots, c_d \in U \right\} \ .$$

In the rest of this section we consider the probability space in which h is selected uniformly at random from $\mathcal{H}_m^d$, $d \geq 1$.

The following fact and corollary were shown by Fredman, Komlós, and Szemerédi [16], and before by Carter and Wegman [8]. (The original proof was only for the case $d = 1$, however the generalization for $d > 1$ is straightforward.)

Fact 2.2 $\mathbf{E}(B_2) \leq n^2/m$.

Corollary 2.3 *The hash function h is injective on S with probability at least $1 - n^2/m$.*

Proof. The function h is injective if and only if $B_2 = 0$. By Fact 2.2 and Markov's inequality, the probability that h is not injective is $\mathbf{Prob}(B_2 \geq 1) \leq n^2/m$. ∎

The following was shown in [11].

Fact 2.4 *If $d \geq 3$ then $B_2 \leq 2n^2/m$ with n^2/m-dominant probability.*

For $r \geq 0$, let A_r be the rth moment of the distribution of s_i,

$$A_r = A_r(h) = \sum_{0 \leq i < m} s_i^r \ . \tag{6}$$

It is easy to see that $A_0 = m$ and $A_1 = n$. Further, it can be shown that if $n = O(m)$ and if h were completely random function, then A_r is linear in n with high probability for all fixed $r \geq 2$. For polynomial hash functions, Dietzfelbinger *et al.* [12] proved the following fact:

Fact 2.5 *Let $r \geq 0$, and $m \geq n$. If $d \geq r$ then there exists a constant $\sigma_r > 0$, depending only on r, such that*

$$\mathbf{Prob}(A_r \leq 2^{\sigma_r} \cdot n) \ \geq \ 1/2 \ .$$

Tighter estimates on the distribution of A_r were given in [11]: (For completeness, the proofs are attached in Appendix B.)

Fact 2.6 *Let $r \geq 2$. If $d \geq r$ then*

$$\mathbf{E}(A_r) \leq n \sum_{1 \leq j \leq r} \left\{ {r \atop j} \right\} \left(\frac{2n}{m}\right)^{j-1} \ ,$$

where $\left\{ {r \atop j} \right\}$ is the Stirling number of the second kind.[1]

Fact 2.7 *Let $\epsilon > 0$ be constant. If $d \geq 2r$ and $m \geq n$ then $A_r \leq (1+\epsilon)\mathbf{E}(A_r)$ with n-dominant probability.*

[1]For $k \geq j \geq 0$, the *Stirling number of the second kind*, $\left\{ {k \atop j} \right\} = \frac{1}{j!} \sum_{i=0}^{k} (-1)^{j-i} \binom{j}{i} i^k$, is the number of ways of partitioning a set of k distinct elements into j nonempty subsets (e.g., [31, Chapter 6]).

3 A Framework for Hashing by Oblivious Execution

3.1 An algorithm template

The input to the algorithm is a set S of n keys, given in an array. The hashing algorithm works in two stages, which correspond to the two level hashing scheme of Fredman, Komlós, and Szemerédi [16].

In the first stage a level-1 hash function f is chosen. This function is selected at random from the class $\mathcal{H}_m^d$, where d is a sufficiently large constant to be selected in the analysis, and $m = \Theta(n)$. The hash function f partitions the input set into m *buckets*; bucket i, $i = 0, \ldots, m-1$, is the set $S \cap f^{-1}(i)$. The first stage is easily implemented in constant time. The main effort is in the implementation of the second stage, which is described next.

The second level of the hash table is built in the second stage of the algorithm. For each bucket a private memory region, called a *block*, is assigned. The address of the memory block allocated to bucket i is recorded in cell i of a designated array `ptr` of size m. Also, for each bucket, a level-2 function is constructed; this function injectively maps the bucket into its block. The descriptions of the level-2 functions are written in `ptr`.

Let us call a bucket *active* if an appropriate level-2 function has not yet been found, and *inactive* otherwise. At the beginning of the stage all buckets are active, and the algorithm terminates when all buckets have become inactive. The second stage consists of $O(\lg \lg n)$ iterations, each executing in constant time. The iterative process rapidly reduces the number of active buckets and the number of active keys.

At each iteration t, a new memory segment is used. This segment is partitioned into M_t blocks of size K_t each, where M_t and K_t will be set in the analysis. Each bucket and each key is associated with one processor. The operation of each active bucket in each iteration is given in Figure 2.

Allocation: The bucket selects at random one of the M_t memory blocks. If the same block was selected by another bucket, then the bucket remains active and does not participate in the next step.

Hashing: The bucket selects at random two functions from $\mathcal{H}_{K_t}^1$, and then tries to hash itself into the block separately by each of these functions. If either one of the functions is injective, then its description and the memory address of the block are written in the appropriate cell of array `ptr` and the bucket becomes inactive. Otherwise, the bucket remains active and carries on to the next iteration.

Figure 2: *The two steps of an iteration, based on oblivious execution.*

In a few of the last iterations, it may become necessary for an iteration to repeat its body more than once, but no more than a constant number of times. The precise conditions and the number of repetitions are given in Section 5.

The hash table constructed by the algorithm supports lookup queries in constant time. Given a key x, a search for it begins by reading the cell $\mathsf{ptr}[f(x)]$. The contents of this cell defines the level-2 function to be used for x as well as the address of the memory block in which x is stored. The actual offset in the block in which x is stored is given by the injective level-2 hash function found in the *Hashing* step above.

3.2 Implementations

The algorithm template described above constitutes a framework for building parallel hashing algorithms. The execution of these algorithms is oblivious in the sense that the iterative process of finding level-2 hash functions does not require information about the number or size of active buckets. Successful termination and performance are dependent on the a priori setting of the parameters d, M_t and K_t. The effectiveness of the allocation step relies on having sufficiently *many* memory blocks; the effectiveness of the hashing step relies on having sufficiently *large* memory blocks. The requirement of keeping the total memory linear imposes a tradeoff between the two parameters. The challenge is in finding a balance between M_t and K_t, so as to achieve a desired rate of decay in the number of active buckets. The number of active keys can be deduced from the number of active buckets based on the characteristics of the level-1 hash function, as determined by d.

We will show two different implementations of the algorithm template, each leading to an analysis of a different nature. The first implementation is given in Section 4. There, the parameters are selected in such a way that in each iteration, the number of active buckets is expected to decrease by a constant factor. Although each iteration may fail with constant probability, there is a geometrically decreasing series which bounds from above the number of active buckets in each iteration. After $O(\lg\lg n)$ iterations, the expected number of active keys and active buckets becomes $n/(\lg n)^{\Omega(1)}$. The remaining keys are hashed in additional constant time using a different approach, after employing an $O(\lg\lg n)$ time procedure.

From a technical point of view, the analysis of this implementation imposes relatively modest requirements on the level-1 hash function, since it only uses first-moment analysis (i.e., Markov's inequality). Moreover, it only requires a simpler version of the hashing step, in which only one hash function from $\mathcal{H}^1_{K_t}$ is being used. The expected running time is $O(\lg\lg n)$, but this running time is guaranteed only with (arbitrary small) constant probability.

The second implementation is given in Section 5. This implementation is characterized by a doubly-exponential rate of decrease[2] in the number of active buckets and keys. After $O(\lg\lg n)$

[2]A sequence $v_0, v_1, \ldots$ decreases in an *exponential* rate if for all t, $v_t \le v_0/(1+\epsilon)^t$ for some $\epsilon > 0$; the sequence decreases in a *doubly-exponential* rate if for all t, $v_t \le v_0/2^{(1+\epsilon)^t}$ for some $\epsilon > 0$.

iterations all keys are hashed without any further processing. This implementation is superior in several other respects: its time performance is with high probability, each key is only handled by its original processor, and it forms a basis for further improvements in reducing the number of random bits.

From a technical point of view, the analysis of this implementation is more subtle and imposes more demanding requirements on the level-1 hash function, since it uses second-moment analysis (i.e., Chebyshev's inequality). Achieving a doubly-exponential rate of decrease required a more careful selection of parameters, and was done using a "symbolic spreadsheet" approach.

Together, these implementations demonstrate two different paradigms for fast parallel randomized algorithms, each involving a different flavor of analysis. One only requires an exponential rate of decrease in problem size, and then relies on reallocation of processors to items. (Subsequent works that use this paradigm and its extensions are mentioned in Section 10.) This paradigm is relatively easy to understand and not too difficult to analyze, using a framework of probabilistic induction and analysis by expectations. The analysis shows that each iteration succeeds with constant probability, and that this implies an overall constant success probability. In contrast, the second implementation shows that each iteration succeeds with n-dominant probability, and that this implies an overall n-dominant success probability. The analysis is significantly more subtle, and relies on more powerful techniques of second moment analysis. The second paradigm consists of a doubly-exponential rate of decrease in the problem size, and hence does not require any wrap-up step.

4 Obtaining Exponential Decrease

This section presents our first implementation of the algorithm template. Using a rather elementary analysis of expectations, we show that at each iteration the problem size decreases by a constant factor with (only) constant probability. The general framework described in Section 4.1 shows that this implies that the problem size decreases at an overall exponential rate.

After $O(\lg\lg n)$ iterations, the number of keys is reduced to $n/(\lg n)^{\Omega(1)}$. A simple load balancing algorithm now allocates $(\lg n)^{\Omega(1)}$ processors to each remaining key. Using the excessive number of processors, each key is finally hashed in constant time.

4.1 Designing by Expectation

Consider an iterative randomized algorithm, in which after each iteration some measure of the problem decreases by a random amount. In a companion paper [22] we showed that at each iteration one can actually assume that in 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".

It is justified by the following lemma.

Lemma 4.1 (probabilistic induction [22]) *Consider an iterative randomized process in which, for all $t \geq 0$, the following holds: iteration $t+1$ succeeds with probability at least $1/2$, provided that among the first t iterations at least $t/4$ were successful. Then, with probability $\Omega(1)$, for every $t > 0$ the number of successful iterations among the first t iterations is at least $t/4$.*

4.2 Parameters setting and analysis

Let the level-1 function be taken from $\mathcal{H}_m^{10}$, i.e., set

$$d = r = 10 \ . \tag{7}$$

Further, set

$$m = 4n \ . \tag{8}$$

Let

$$K_t = 2^{4 + \sigma_{10}/5 + t/5} \ , \tag{9}$$

$$M_t = m\, 2^{4 - t/4} \ , \tag{10}$$

where σ_{10} is as in Fact 2.5.

To simplify the analysis, we allow the parameters K_t and M_t to assume non-integral values. In actual implementation, they must be rounded up to the nearest integer. This does not increase memory requirements by more than a constant factor; all other performance measures can only be improved.

Memory usage The memory space used is

$$\sum_t M_t K_t = m 2^{8+\sigma_{10}/5} \sum_t 2^{t/5 - t/4} = O(n) \ .$$

Lemma 4.2 *Let v_t be the number of active buckets at the beginning of iteration t. Then,*

$$\textbf{Prob}\left(\forall\, t \geq 0 \quad v_t \leq m 2^{-t/4}\right) = \Omega(1).$$

Proof. We assume that the level-1 function f satisfies

$$A_{10} \leq 2^{\sigma_{10}} \cdot m \ . \tag{11}$$

By Fact 2.5, (11) holds with probability at least 1/2.

The proof is by continued by using Lemma 4.1. Iteration t is *successful* if $v_{t+1} \leq v_t/2$. Thus, the number of active buckets after j successful iterations is at most $m2^{-j}$.

The probabilistic inductive hypothesis is that among the first t iterations at least $t/4$ were successful, that is

$$v_t \leq m2^{-t/4} \ . \tag{12}$$

The probabilistic inductive step is to show that

$$\mathbf{Prob}\,(v_{t+1} \leq v_t/2) \geq 1/2 \ .$$

In each iteration the parameters K_t and M_t were chosen so as to achieve constant deactivation probability for buckets of size at most

$$\beta_t = \sqrt{K_t/8} = 2^{(1+\sigma/5+t/5)/2}. \tag{13}$$

We distinguish between the following three types of events, "failures", which may cause a bucket to remain active at the end of an iteration.

(i) *Allocation Failure.* The bucket may select a memory block which is also selected by other buckets.

Let $\rho_1(t)$ be the probability that a fixed bucket does not successfully reserve a block in the allocation step. Since there are at most v_t buckets, each selecting at random one of M_t memory blocks, $\rho_1(t) \leq v_t/M_t$. By (12) and (10)

$$\rho_1(t) \leq m2^{-t/4}/m2^{4-t/4} = 1/16 \ .$$

(ii) *Size Failure.* The bucket may be too large for the current memory block size. As a result, the probability for it to find a level-2 hash function is not high enough.

Let v'_t be the number of buckets at the beginning of iteration t that are larger than β_t. By (11),

$$v'_t \cdot \beta_t^{10} \leq A_{10} \leq 2^{\sigma_{10}} m.$$

Therefore, by (13),

$$v'_t \leq 2^{\sigma_{10}} m/\beta_t^{10} = m2^{\sigma_{10}-5(1+\sigma_{10}/5+t/5)} = m2^{-5-t}. \tag{14}$$

Without loss of generality, we assume that if $v_{t+1} \leq v_t/2$ then $v_{t+1} = v_t/2$ (i.e., if more buckets that are needed become inactive, then some of them are still considered as active). Thus, for the purpose of analysis,

$$v_t \geq m2^{-t}. \tag{15}$$

We have then

$$v'_t \leq v_t 2^{-5} < v_t/16 \ .$$

(iii) *Hash Failure.* A bucket may fail to find an injective level-2 hash function even though it is sufficiently small and it has uniquely selected a block.

Let $\rho_3(t)$ be the probability that a bucket of size at most β_t is not successfully mapped into a block of size K_t in the hashing step. By Corollary 2.3 and (13)

$$\rho_3(t) \leq \beta_t^2/K_t = 1/8 \ .$$

A bucket of size at most β_t that successfully reserves a block of size K_t, and that is successfully mapped into it, becomes inactive. The expected number of active buckets at the beginning of iteration $t+1$ can therefore be bounded by

$$\mathbf{E}\left(v_{t+1}\right) \leq v_t\rho_1(t) + v'_t + v_t\rho_3(t) \leq v_t\left(1/16 + 1/16 + 1/8\right) \leq v_t/4 \ .$$

By Markov's inequality

$$\mathbf{Prob}\left(v_{t+1} \leq v_t/2\right) \geq 1/2 \ ,$$

proving the inductive step. The lemma follows. ∎

Lemma 4.3 *Let n_t be the number of active keys at the beginning of iteration t. Then*

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

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

Proof. It follows from (11), by using a simple convexity argument, that n_t is maximal when all active buckets at the beginning of iteration t are of the same size q_t. In this case, by (11),

$$v_t \cdot q_t^{10} \leq 2^{\sigma_{10}} \cdot m$$

and

$$n_t = v_t \cdot q_t \leq v_t \cdot \left(\frac{2^{\sigma_{10}} m}{v_t}\right)^{0.1} = (2^{\sigma_{10}} m)^{0.1} v_t^{0.9}.$$

Therefore, by Lemma 4.2, the lemma follows. ∎

By Lemma 4.3 and Lemma 4.2 we have an exponential decrease in the number of active keys and in the number of active buckets with probability $\Omega(1)$. The number of active keys becomes $n/(\lg n)^c$, for any constant $c > 0$, after $O(\lg\lg n)$ iterations with probability $\Omega(1)$.

4.3 A final stage

After the execution of the second stage with the parameter setting as described above, the number of available resources (memory cells and processors) is a factor of $(\lg n)^{\Omega(1)}$ larger than the number of active keys. This resource redundancy makes it possible to hash the remaining active keys in constant time, as described in the remainder of this section.

All keys that were not hashed in the iterative process will be hashed into an auxiliary hash table of size $O(n)$. Consequently, the implementation of a lookup query will consist of searching the key in both hash tables.

The auxiliary hash table is built using the the 2-level hashing scheme. A level-1 function maps the set of active keys into an array of size n. This function is selected at random from a class of hash functions presented by Dietzfelbinger and Meyer auf der Heide [14, Definition 4.1]. It has the property that with n-dominant probability each bucket is of size smaller than $\lg n$ [14, Theorem 4.6(b')]. For the remainder of this section we assume that this event indeed occurs. (Alternatively, we can use the n^{ϵ}-universal class of hash functions presented by Siegel [43].)

Each active key is allocated $2\lg n$ processors, and each active bucket is allocated $4(\lg n)^3$ memory. The allocation is done by mapping the active keys injectively into an array of size $O(n/\lg n)$, and by mapping the indices of buckets injectively into an array of size $O(n/(\lg n)^3)$. These mappings can be done in $O(\lg\lg n)$ time with n-dominant probability, by using the simple renaming algorithm from [20].

The remaining steps take constant time. We independently select $2\lg n$ linear hash functions and store them in a designated array. These hash functions will be used by all buckets.

The memory allocated to each bucket is partitioned into $2\lg n$ memory blocks, each of size $2\lg^2 n$. Each bucket is mapped in parallel into its $2\lg n$ blocks by the $2\lg n$ selected linear hash functions, and each mapping is tested for injectiveness. This is carried out by the $2\lg n$ processors allocated to each key. For each bucket, one of the injective mappings is selected as a level-2 function. The selection is made by using the simple 'leftmost 1' algorithm of [15].

If for any of the buckets all the mappings are not injective then the construction of the auxiliary hash table fails.

Lemma 4.4 *Assume that the number of keys that remain active after the iterative process is at most $n/(\lg n)^3$. Then, the construction of the auxiliary hash table succeeds with n-dominant probability.*

Proof. Recall that each bucket is of size at most $\lg n$; A mapping of a bucket into its memory block of size $2(\lg n)^2$ is injective with probability at least $1/2$ by Corollary 2.3. The probability that a bucket has no injective mapping is therefore at most $1/n^2$. With probability at least $1-1/n$, every bucket has at least one injective mapping. ∎

It is easy to identify failure. If the algorithm fails to terminate within a designated time, it can be restarted. The hash table will be therefore always constructed. Since the overall failure probability is constant, the expected running time is $O(\lg \lg n)$.

5 Obtaining Doubly-Exponential Decrease

The implementation of the algorithm template that was presented in the previous section maintains an exponential decrease in the number of active buckets throughout the iterations. This section presents the implementation in which the number of active buckets decreases at a doubly-exponential rate.

Intuitively, the stochastic process behind the algorithm template has a potential for achieving doubly-exponential rate: If a memory block is sufficiently large in comparison to the bucket size then the probability of the bucket to remain active is inversely proportional to the size of the memory block (Corollary 2.3). Consider an idealized situation in which this is the case. If at iteration t there are m_t active buckets, each allocated a memory block of size K_t, then at iteration $t+1$ there will be m_t/K_t active buckets, and each of those could be allocated a memory block of size K_t^2; at iteration $t+2$ there will be m_t/K_t^3 active buckets, each to be allocated a memory block of size K_t^4, and so on.

In a less idealized setting, some buckets do not deactivate because they are too large for the current value of K_t. The number of such buckets can be bounded above by using properties of the level-1 hash function. It must be guaranteed that the fraction of "large buckets" also decreases at a doubly-exponential rate.

The illustrative crude calculation given above assumes that memory can be evenly distributed between the active buckets. To make the doubly-exponential rate possible, the failure probability of the allocation step, and hence the ratio m_t/M_t, must also decrease at a doubly-exponential rate.

Establishing a bound on the number of "large blocks" and showing that a large fraction of the buckets are allocated memory blocks were also of concern in the previous section. There, however, it was enough to show constant bounds on the probabilities of *allocation failure*, *size failure* and *hash failure*.

The parameter setting which establishes the balance required for the doubly-exponential rate is now presented. Following that is the analysis of the algorithm performance. The section concludes with a description of how the parameters were selected.

5.1 Parameters setting

Let the level-1 function be taken from $\mathcal{H}_m^{18}$, i.e., set $d = 18$. Let

$$r = 9 \ . \tag{16}$$

Further, set

$$m = 160n \ . \tag{17}$$

Let

$$K_t = 2^{a\lambda^t + b_1 t + c_1} \tag{18}$$

$$M_t = n \ 2^{-a\lambda^t - b_2 t + c_2} \ , \tag{19}$$

where

$$\lambda = 18/13 \ ; \quad a = 8/13 \ ; \quad b_1 = 1/5 \ ; \quad b_2 = 9/20 \ ; \quad c_1 = 73/25 \ ; \quad c_2 = 89/20 \ . \tag{20}$$

5.2 Memory usage

Proposition 5.1 *The total memory used by the algorithm is $O(n)$.*

Proof. By (17), the memory used in the first stage is $O(n)$. The memory used in an iteration t of the second stage is

$$M_t \cdot K_t = n \ 2^{(b_1 - b_2)t + c_1 + c_2} = n \ 2^{-t/4 + 737/100} \ . \tag{21}$$

The total memory used by the second stage is therefore at most

$$\sum_{t=0}^{\infty} n \ 2^{-t/4 + 737/100} = \frac{2^{7.37}}{1 - 1/\sqrt[4]{2}} \cdot n = O(n) \ . \tag{22}$$

∎

5.3 Framework for time performance analysis

Let m_t be defined by

$$m_t = n \ 2^{-\lambda^t - b_2 t + 1} \ . \tag{23}$$

The run-time analysis of the second stage is carried out by showing:

Lemma 5.2 *With n-dominant probability, the number of active buckets in the beginning of iteration t is at most m_t.*

The lemma is proved by induction on t, for $t \leq \lg\lg n / \lg\lambda$. The induction base follows from $m_0 = n$ and the fact that there are at most n active buckets.

In the subsequent subsections, we prove the inductive step by deriving estimates on the number of failing buckets in iteration t under the assumption that at the beginning of the iteration there are at most m_t active buckets. Specifically, we show by induction on t that, with n-dominant probability, the number of active buckets at the end of iteration t is at most

$$m_{t+1} = n\ 2^{-\lambda^{t+1} - b_2(t+1)+1} \ . \tag{24}$$

The bucket may fail to find an injective level-2 hash function. In estimating the number of buckets that fail to find an injective level-2 function during an iteration we assume that the bucket uniquely selected a memory block and that the bucket size is not too large relatively to the current block size. Accordingly, as in Section 4.2, we distinguish between the following three types of events, "failures", which may cause a bucket to remain active at the end of an iteration.

(i) *Allocation Failure.* The bucket may select a memory block which is also selected by other buckets.

(ii) *Size Failure.* The bucket may be too large for the current memory block size. As a result, the probability for it to find a level-2 hash function is not high enough.

(iii) *Hash Failure.* A bucket may fail to find a level-2 hash function even though it is sufficiently small and it has uniquely selected a block.

We will provide estimates for the number of buckets that remain active due to either of the above reasons: in Lemma 5.5 for case (i), in Lemma 5.6 for case (ii), and in Lemma 5.7 and Lemma 5.8 for case (iii). The estimates are all shown to hold with n-dominant probability. The induction step follows from adding all these estimates.

To wrap up, let $t = \lg\lg n / \lg\lambda$. Then, by (23),

$$m_t = n\ 2^{-\lambda^t - b_2 t + 1} = n\ 2^{-\lg n - b_2 t + 1} < 1 \ .$$

We can therefore infer:

Proposition 5.3 *With n-dominant probability, the number of iterations required to deactivate all buckets is at most $\lg\lg n / \lg\lambda$.*

5.4 Failures in Uniquely Selecting a Block

Lemma 5.4 *Let ϵ be fixed, $0 < \epsilon < 1/6$, and suppose that either $m_t > M_t^{1/2+\epsilon}$ or $m_t < M_t^{1/2-\epsilon}$. Let ω be the random variable representing the number of buckets that fail to uniquely select a block. Then, $\omega \leq 2m_t^2/M_t$, with M_t-dominant probability.*

Proof. A bucket has a probability of at most m_t/M_t to have other buckets select the memory block it selected. Therefore,

$$\mathbf{E}\left(\omega\right) \leq m_t^2/M_t \ . \tag{25}$$

Further, ω is stochastically smaller than a binomially distributed random variable ϖ obtained by performing m_t independent trials, each with probability m_t/M_t of success. That is to say, $\mathbf{Prob}\left(\omega \geq \omega_0\right) \leq \mathbf{Prob}\left(\varpi \geq \omega_0\right)$ for all ω_0. Note that $\mathbf{E}\left(\varpi\right) = m_t^2$. If $m_t > M_t^{1/2+\epsilon}$ then

$$\begin{array}{rcl} \mathbf{Prob}\left(\omega > 2m_t^2/M_t\right) & \leq & \mathbf{Prob}\left(\varpi > 2m_t^2/M_t\right) \\ & \overset{\text{by (3)}}{=} & e^{-\Omega(\mathbf{E}(\varpi))} \\ & = & e^{-\Omega(m_t^2/M_t)} \\ & = & e^{-\Omega(M_t^{2\epsilon})} \\ & = & M_t^{-\Omega(1)} \ . \end{array} \tag{26}$$

Otherwise, $m_t < M_t^{1/2-\epsilon}$ and we are in the situation where $\mathbf{E}\left(\omega\right) \ll 1$. Since ω is integer valued and $2m_t^2 > 0$

$$\begin{array}{rcl} \mathbf{Prob}\left(\omega > 2m_t^2/M_t\right) & \leq & \mathbf{Prob}\left(\omega \geq 1\right) \\ & \overset{\text{by (1)}}{\leq} & \mathbf{E}\left(\omega\right) \\ & \overset{\text{by (25)}}{\leq} & m_t^2/M_t \\ & < & M_t^{-2\epsilon} \ . \end{array} \tag{27}$$

∎

The setting not covered by the above lemma is $M_t^{1/2-\epsilon} < m_t < M_t^{1/2+\epsilon}$. This only occurs in a constant number of iterations throughout the algorithm and requires the following special treatment. The body of these iterations is repeated, thus providing a second allocation attempt of buckets that failed to uniquely select a memory block in the first trial.

Let ω_1 and ω_2 be the random variables representing the number of buckets that fail to uniquely select a block in the first and second attempts respectively.

$$\begin{aligned}
\mathbf{Prob}\left(\omega_1 > M_t^{1/2-\epsilon}\right) &\overset{\text{by (1)}}{<} \mathbf{E}(\omega_1)/M_t^{1/2-\epsilon} \\
&\overset{\text{by (25)}}{=} m_t^2/M_t M_t^{1/2-\epsilon} \\
&= m_t^2/M_t^{3/2-\epsilon} \\
&\leq M_t^{1/2+\epsilon}/M_t^{3/2-\epsilon} \\
&= M_t^{3\epsilon-1/2} \\
&= M_t^{-\Omega(1)} \ .
\end{aligned} \tag{28}$$

Therefore, with M_t-dominant probability the second attempt falls within the conditions of Equation (27) and hence $\omega_2 = 0$ with M_t-dominant probability.

Lemma 5.5 *Let $t \leq \lg\lg n/\lg\lambda$. The number of buckets that fail to uniquely select a block is, with n-dominant probability, at most $m_{t+1}/4$.*

Proof. By Lemma 5.4, the number of buckets that fail to uniquely select a memory block is, with M_t-dominant probability, at most

$$\begin{aligned}
2m_t^2/M_t &\overset{\text{by (19),(23)}}{=} 2n^2\ 2^{-2\lambda^t-2b_2t+2}/n\ 2^{-a\lambda^t-b_2t+c_2} \\
&= n\ 2^{(a-2)\lambda^t-b_2t+3-c_2} \\
&\overset{\text{by (20)}}{=} n\ 2^{(8/13-2)\lambda^t-b_2(t+1)+b_2+3-c_2} \\
&= n\ 2^{-(18/13)\lambda^t-b_2(t+1)+9/20+3-89/20} \\
&\overset{\text{by (20)}}{=} n\ 2^{-\lambda^{t+1}-b_2(t+1)-1} \\
&\overset{\text{by (24)}}{=} m_{t+1}/4 \ .
\end{aligned} \tag{29}$$

The above holds also with n-dominant probability since

$$\begin{aligned}
M_t &\overset{\text{by (19)}}{=} n\ 2^{-a\lambda^t-b_2t+c_2} \\
&\geq n\ 2^{-a\lg n-b_2t+c_2} \\
&= n^{1-a}2^{-b_2\lg\lg n/\lg\lambda+c_2} \\
&\overset{\text{by (20)}}{=} n^{5/13}\lg n^{-\Omega(1)} \ .
\end{aligned} \tag{30}$$

■

5.5 Failures in Hashing

In considering buckets which uniquely selected a block which fail to find an injective level-2 function we draw special attention to buckets of size at most

$$\beta_t = \sqrt[4]{K_t/2} \ . \tag{31}$$

Lemma 5.6 *The number of buckets larger than β_t is, with n-dominant probability, at most $m_{t+1}/4$.*

Proof. Let $\mu = m/n = 160$. By incorporating the appropriate values for the Stirling numbers of the second kind into Fact 2.6, we get

$$\begin{aligned} \mathbf{E}\,(A_r) &\leq \left(1 + \frac{510}{\mu} + \frac{12100}{\mu^2} + \frac{62160}{\mu^3} + \frac{111216}{\mu^4} + \frac{84672}{\mu^5} + \frac{29568}{\mu^6} + \frac{4608}{\mu^7} + \frac{256}{\mu^8}\right) n \\ &\overset{\text{by (17)}}{\leq} 4.6756\, n \ . \end{aligned}$$

Therefore, by Fact 2.7, with n-dominant probability

$$A_r \leq 6n \ . \tag{32}$$

From the above and (6) it follows that the number of buckets bigger than β_t is, with n-dominant probability, at most

$$\begin{aligned} 6n/\beta_t^r &\overset{\text{by (31),(16)}}{=} 6n/(K_t/2)^{9/4} \\ &\overset{\text{by (18)}}{=} 6n\ 2^{-9(a\lambda^t + b_1 t + c_1 - 1)/4} \\ &\overset{\text{by (20)}}{=} 6n\ 2^{-(18/13)\lambda^t - (9/20)t - 108/25} \\ &\overset{\text{by (20)}}{=} 6n\ 2^{-\lambda^{t+1} - b_2(t+1) + b_2 - 108/25} \\ &\overset{\text{by (20)}}{=} 6n\ 2^{-\lambda^{t+1} - b_2(t+1) + 9/20 - 108/25} \\ &= 6n\ 2^{-\lambda^{t+1} - b_2(t+1) - 387/100} \\ &= n\ 2^{-\lambda^{t+1} - b_2(t+1) + 1 + (\lg 3 - 387/100)} \\ &\overset{\text{by (24)}}{=} m_{t+1} 2^{\lg 3 - 387/100} \\ &= m_{t+1} 2^{-2.285\ldots} \\ &< m_{t+1}/4 \ . \end{aligned} \tag{33}$$

∎

The analysis of hashing failures of buckets that are small enough is further split into two cases.

Lemma 5.7 *Suppose that $m_t/2K_t \geq \sqrt{n}$. Then the number of buckets of size at most β_t that fail in the hashing step of the iteration is, with n-dominant probability, at most $m_{t+1}/4$.*

Proof. Without loss of generality, we may assume that there are exactly m_t active buckets of size at most β_t that participate in Step 2. When such a bucket is mapped into a memory block of size K_t, the probability of the mapping being non-injective is, by Corollary 2.3, at most $\beta_t^2/K_t = 1/\sqrt{2K_t}$. The probability that the bucket fails in both hashing attempts is therefore at most $1/2K_t$. Let

$\tilde{m}_t$ be the total number of such failing buckets. Then, $\mathbf{E}(\tilde{m}_t) \leq m_t/2K_t$. By Fact 2.1, with $m_t/2K_t$-dominant probability,

$$
\begin{array}{rcl}
\tilde{m}_t & \leq & 2(m_t/2K_t) \\
& = & m_t/K_t \\
& \stackrel{\text{by (18),(23)}}{=} & n\ 2^{-\lambda^t - b_2 t + 1 - a\lambda^t - b_1 t - c_1} \\
& = & n\ 2^{-(1+a)\lambda^t - (b_2+b_1)t + 1 - c_1} \\
& \stackrel{\text{by (20)}}{\leq} & n\ 2^{-(21/13)\lambda^t - b_2(t+1) + 1 + b_2 - c_1} \\
& \stackrel{\text{by (20)}}{\leq} & n\ 2^{-\lambda^{t+1} - b_2(t+1) + 1 + b_2 - c_1} \\
& \stackrel{\text{by (20),(24)}}{=} & m_{t+1}\ 2^{9/20 - 73/25} \\
& < & m_{t+1}/4\ .
\end{array}
\tag{34}
$$

Note that since $m_t/2K_t \geq \sqrt{n}$, the above holds with n-dominant probability and we are done. ∎

Lemma 5.8 *Suppose that $m_t/2K_t < \sqrt{n}$. Then, by repeating the hashing step of the iteration a constant number of times, we get $\tilde{m}_t = 0$, with n-dominant probability.*

Proof. We have

$$
n\ 2^{-(1+a)\lambda^t - (b_2+b_1)t - c_1} \stackrel{\text{by (18),(23)}}{=} m_t/2K_t < \sqrt{n}\ , \tag{35}
$$

and thus,

$$
2^{(1+a)\lambda^t + (b_2+b_1)t + c_1} > \sqrt{n}\ . \tag{36}
$$

Therefore,

$$
\begin{array}{rcl}
K_t & \stackrel{\text{by (18)}}{=} & 2^{a\lambda^t + b_1 t + c_1} \\
& \geq & 2^{\delta\left((1+a)\lambda^t + (b_2+b_1)t + c_1\right)} \\
& \stackrel{\text{by (36)}}{>} & n^{\delta/2}\ ,
\end{array}
\tag{37}
$$

for some constant $\delta > 0$. Recall from the proof of Lemma 5.7 that a bucket fails in the hashing step with probability at most $1/2K_t$. By (37), if the iteration body is repeated $\lceil 2/\delta \rceil + 1$ times, the failure probability of each bucket becomes at most $(2K_t)^{-2/\delta - 1} < 2^{-2/\delta}/2K_t n$, and

$$
\mathbf{E}(\tilde{m}_t) < m_t 2^{-2/\delta}/2K_t n < 2^{-2/\delta}\sqrt{n}/n = 2^{-2/\delta}/\sqrt{n}\ .
$$

The lemma follows by Markov inequality. ∎

6 Reducing the Number of Random Bits

In this section we show how to reduce the number of random bits used by the hashing algorithm.

The algorithm as described in the previous section consumes $\Theta(n \lg u)$ random bits, where $u = |U|$: the first iteration already uses $\Theta(n \lg u)$ random bits; for each subsequent iteration, the number of random words from U which are used is by at most a constant factor larger than the memory used in that iteration, resulting in a total of $\Theta(n \lg u)$ random bits.

The sequential hashing algorithm of Fredman, Komlós, and Szemerédi [16] can be implemented with only $O(\lg \lg U + \lg n)$ random bits [11]. We show how the parallel hashing algorithm can be implemented with $O(\lg \lg U + \lg n \lg \lg n)$ random bits.

We first show how the algorithm can be modified so as to reduce the number of random bits to $O(\lg u \lg \lg n)$. The first stage requires $O(1)$ random elements from U for the construction of the level-1 function, and remains unchanged. An iteration t of the second stage required $O(m_t)$ random elements from U; it is modified as follows.

Allocation step If each bucket independently selects a random memory block then $O(m_t \lg M_t)$ random bits are consumed. This can be reduced to $O(\lg m)$ by making use of *polynomial hash functions*:

Lemma 6.1 *Using $6 \lg m$ random bits, a set $R \subseteq [0, m-1]$ of size m_t can be mapped in constant time into an array of size $3M_t$ such that the number of colliding elements is at most $2m_t^2/M_t$, with M_t-dominant probability.*

Proof. Let $h_1 \in \mathcal{H}^3_{2M_t}$ and $h_2 \in \mathcal{H}^1_{M_t}$ be selected at random. Then, the image of a bucket i is defined by

$$g_t(i) = \begin{cases} h_1(i) & \text{if } \not\exists j \in R,\, j \neq i,\, h_1(i) = h_1(j) \\ 2M_t + h_2(i) & \text{otherwise} \end{cases} . \tag{38}$$

Algorithmically, h_1 is first applied to all elements and then h_2 is applied to the elements which collided under h_1. The colliding elements of g_t are those which collided both under h_1 and under h_2.

Let R' be the set of elements that collide under h_1. Clearly, $|R'| \leq 2B_2(h_1)$. Let ϵ be some constant, $0 < \epsilon < 1/6$. Consider the following three cases:

1. $m_t \leq M_t^{1/2-\epsilon}$:

   By Corollary 2.3, $\textbf{Prob}\,(R' \neq \emptyset) \leq m_t^2/2M_t \leq M_t^{-2\epsilon}/2$.

2. $m_t \geq M_t^{1/2+\epsilon}$:

   It follows from Fact 2.4 that $B_2 \leq 2m_t^2/2M_t = m_t^2/M_t$ with $m_t^2/2M_t$-dominant probability. As $|R'| \leq 2B_2$ and $m_t^2/2M_t \geq M_t^{2\epsilon}/2$ we have that $|R'| \leq 2m_t^2/M_t$ with M_t-dominant probability.

3. $M_t^{1/2-\epsilon} < m_t < M_t^{1/2+\epsilon}$:

   By Fact 2.2, $\mathbf{E}(B_2(h_1)) \leq m_t^2/2M_t < M_t^{2\epsilon}/2$ and by Markov's inequality,

$$\mathbf{Prob}\left(B_2(h_1) > M_t^{1/2-\epsilon}/2\right) \leq M_t^{2\epsilon}/M_t^{1/2-\epsilon} = M_t^{3\epsilon-1/2} \ .$$

   Therefore, with M_t-dominant probability, $|R'| \leq 2B_2(h_1) \leq M_t^{1/2-\epsilon}$, in which case, by Corollary 2.3,

$$\mathbf{Prob}(h_2 \text{ is not injective over } R') \leq |R'|^2/M_t \leq M_t^{-2\epsilon} \ .$$

■

Invoking the above procedure for block allocation does not increase the total memory consumption of the algorithm by more than a constant factor.

Hashing step The implementation of the hashing part of the iteration body using *independent* hash functions for each of the active buckets consumes $O(m_t \lg u)$ random bits. This can be reduced to $O(\lg u)$ by using hash functions which are only *pairwise independent*. This technique and its application in the context of hash functions are essentially due to [10, 11].

The modification to the step is as follows. In each hashing attempt executed during the step, four global parameters $a_0, a_1, b_0, b_1 \in U$ are selected at random by the algorithm. The hash function attempted by a bucket i is

$$h_i(x) := ((c_0(i) + c_1(i)x) \bmod u) \bmod K_t \tag{39}$$

where

$$\begin{aligned} c_0(i) &= (ia_0 + b_0) \bmod u \\ c_1(i) &= (ia_1 + b_1) \bmod u \ . \end{aligned}$$

All hashing attempts of the same bucket are fully independent. Thus, the proof of Lemma 5.8 is unaffected by this modification. Recall that Fact 2.1 assumes only pairwise independence. Since h_i, $i = 0, \ldots, m-1$, are pairwise independent, the proof of Lemma 5.7 remains valid as well.

The above leads to a reduction in the number of random bits used by the algorithm to $O(\lg u \lg\lg n)$.

The number of random bits can be further reduced as follows: Employ a pre-processing hashing step in which the input set S is injectively mapped into the range $[0, n^3 - 1]$. This is done by applying a hash function π selected from an appropriate class, to map the universe U into this range. Then the algorithm described above is used to build a hash table for the set $\pi(S)$. A lookup of a key x is done by searching for $\pi(x)$ in this hash table.

The simple class of hash functions $\mathcal{H}_m^3$ is appropriate for this universe reduction application. It was shown in [11] that the class $\mathcal{H}_m^3$ has the following properties:

1. A selection of a random function π from the class requires $O(\lg \lg u + \lg n)$ random bits.

2. A selection can be made in constant time by a single processor.

3. The function π is injective over S with n-dominant probability.

4. Computing $\pi(x)$ for any $x \in U$ can be done in constant time.

This pre-processing is tantamount to a reduction in the size of the universe, after which application of the algorithm requires only $O(\lg n \lg \lg n)$ bits. The total number of random bits used is therefore

$$O(\lg \lg u + \lg n \lg \lg n) \ .$$

7 Obtaining Optimal Speedup

The description of the algorithm in Section 3 assumed that the number of processors is n; thus the time-processor product is $O(n \lg \lg n)$. Our objective in this section is a work-optimal implementation where this product is $O(n)$, and p, the number of processors, is maximized.

When $p < n$, the key array and the bucket array are divided into p *sectors*, one per processor. A parallel step of the algorithm is executed by having each processor traverse its sector and execute the tasks included in it.

A key is *active* if its bucket is active. Let n_t be the number of active keys in the beginning of iteration t. Assume that the implemented algorithm has reached the point where $n_t = O(n/\lg \lg n)$. Further assume that these active elements are gathered in an array of size $O(n/\lg \lg n)$. Then, applying the non-optimal algorithm of Section 3 with $p \leq n/\lg \lg n$, and each processor being responsible for $n/p \lg \lg n$ problem instances, gives a running time of

$$O\left(\frac{n}{p \lg \lg n} \lg \lg \left(\frac{n}{\lg \lg n}\right)\right) = O\left(n/p\right)$$

which is work-optimal.

We first show that the problem size is reduced sufficiently for the application of the non-optimal algorithm after $O(\lg \lg \lg \lg n)$ iterations.

Lemma 7.1 *There exists $t_0 = O(\lg \lg \lg \lg n)$ such that $n_{t_0} = O(n/\lg \lg n)$ with n-dominant probability.*

Proof. The number of active buckets decreases at a doubly-exponential rate as can be seen from Lemma 5.2. To see that the number of keys decreases at a doubly-exponential rate as well, we show that with n-dominant probability

$$n_t \leq 1.23n2^{-2a\lambda^{t-1}-2b_1t+8/9} \ . \tag{40}$$

Inequality (32), $A_r \leq 6n$, clearly holds when the summation is over active buckets only. By a convexity argument, the total number of keys in active buckets is maximized when all active buckets are of equal size. The number of active buckets is bounded from above by m_t. Therefore,

$$n_t \leq (6n)^{1/r} m_t^{1-1/r} \quad . \tag{41}$$

Inequality (40) is obtained from (41) by replacing in m_t by its definition in (23) and then substituting numerical values for the parameters using (16) and (20).

The lemma follows by choosing an appropriate value for t_0 with respect to (23) and (40). ∎

It remains to exhibit a work-efficient implementation of the first t_0 steps of the algorithm. This implementation outputs the active elements gathered in an array of size $O(n/\lg\lg n)$. The rest of this section is dedicated to the description of this implementation.

As the algorithm progresses, the number of active keys and the number of active buckets decrease. However, the decrease in the number of active elements in different sectors is not necessarily identical. The time of implementing one parallel step is proportional to the number of active elements in the largest sector. It is therefore crucial to occasionally balance the number of active elements among different sectors in order to obtain work efficiency.

Let the *load* of a sector be the number of active elements (tasks) in it. A *load balancing* algorithm takes as input a set of tasks arbitrarily distributed among p sectors; using p processors it redistributes this set so that the load of each sector is greater than the average load by at most a constant factor. Suppose that we have a load balancing algorithm whose running time, using p processors, is $T_{\mathrm{lb}}(p)$ with n-dominant probability. If load balancing is applied after step t then the size of each sector is $O(n_t/p)$.

We describe a simple work-optimal implementation in which load balancing is applied after each of the first t_0 parallel steps. A parallel step t executes in time which is in the order of

$$\frac{n_t}{p} + T_{\mathrm{lb}}(p) \quad .$$

The total time of this implementation is in the order of

$$\sum_{t=1}^{t_0} \left(\frac{n_t}{p} + T_{\mathrm{lb}}(p) \right)$$

Since n_t decreases at least at an exponential rate, the total time is in the order of

$$\frac{n}{p} + t_0 \, T_{\mathrm{lb}}(p)$$

which is $O(n/p)$ for

$$p = O\left(\frac{n}{T_{\mathrm{lb}}(p) \lg\lg\lg\lg n} \right) \quad .$$

Using the load balancing algorithm of [20] which runs in $T_{\rm lb}(p) = O(\lg\lg p)$ time, we conclude that with n-dominant probability the running time on a p-processor machine is

$$O(n/p + \lg\lg p \lg\lg\lg\lg n) \ .$$

The load balancing algorithm applied consumes $O(p \lg\lg p)$ random bits. All these bits are used in a random mapping step which is very similar to the allocation step of the hashing algorithm. Thus, by a similar approach as the mapping procedure in Lemma 6.1 it may be established that the number of random bits in the load balancing algorithm can be reduced to $O(\lg p \lg\lg p)$.

We finally remark that using load balancing in a more efficient, yet as simple way, as describe in [23], yields a faster work-efficient implementation. The technique is based on carefully choosing the appropriate times for invoking the load balancing procedure; it applies to any algorithm in which the problem size has an exponential rate of decrease, and it hence applies to the implementation of Section 4 as well. In such an implementation the load balancing algorithm is only used $O(\lg^* n)$ times, resulting in a parallel hashing algorithm that takes $O(n/p + \lg\lg n \lg^* n)$ time with n-dominant probability.

8 Model of Computation

In this section we give a closer attention to the details of the implementation on a PRAM, and study the type of concurrent memory access required by our algorithm. We first present an implementation on COLLISION, and its extension to the weaker TOLERANT model. We proceed by presenting an implementation on the even weaker ROBUST model. The hash-table constructed in this implementation only supports searches in $O(\lg\lg n)$ time. Finally, we examine the concurrent read capability needed by the implementations.

8.1 Implementation on COLLISION and on TOLERANT

We describe an implementation on COLLISION. This implementation is also valid for TOLERANT, since each step of COLLISION can be simulated in constant time on TOLERANT provided that, as it is the case here, only linear memory is used [32].

Initialization The selection of the level-1 hash function is done by a single processor. Since the level-1 function is a polynomial of a constant degree, its selection can be done by a single processor and be read by all processors in constant time, using a singe memory cell of $\lceil \max\{\lg\lg u, \lg n\} \rceil$ bits. No concurrent-write operation is required for the implementation of this stage.

Bucket representatives The algorithm template assumes that each bucket can act as a single entity for some operations, e.g., selecting a random block and selecting a random hash function. Since usually several keys belong to the same bucket, it is necessary to coordinate the actions of the processors allocated to these keys. A simple way of doing so is based on the fact that there are only linearly many buckets and that a bucket is uniquely indexed by the value of f, the level-1 hash function, on its members. A processor whose index is determined by the bucket index acts as the *bucket representative* and performs the actions prescribed by the algorithm to the bucket.

Allocation and Hashing steps A processor representing an active bucket selects a memory block and a level-2 hash function, and records these selections in a designated cell. All processors with keys in that bucket read then that cell and use the selected block in the hashing step. Each participating processor (whose key belongs in an active bucket) writes its key in the cell determined by its level-2 hash function, and examines the cell contents to see if the write operation was successful. A processor for which the write failed will then attempt to write its key to position i of array ptr, where i is the number of the bucket this processor belongs to. Processors belonging to bucket i can then learn if the level-2 function selected for their bucket is injective by reading the content of $\mathsf{ptr}[i]$. A change in value or a collision symbol indicate non-injectiveness. To complete the process, the array ptr is restored for the next hashing attempt. This restoration can be done in constant time since this array is of linear size.

In summary we have

Proposition 8.1 *The algorithms of Theorem 1 can be implemented on* TOLERANT.

8.2 Implementation on ROBUST

We now describe an implementation that, at the expense of slowing down the lookup operation, makes no assumption about the result of a concurrent-write into a cell. Specifically, we present an implementation on the ROBUST model, for which a lookup query may take $O(\lg\lg n)$ time in the worst case, but $O(1)$ expected time for keys in the table.

The difficulty with the ROBUST model is in letting all processors in a bucket know whether the level-2 hash function of their bucket is injective or not. The main idea in the modified implementation is in allowing iterations to proceed without determining whether level-2 hash functions are injective or not; whenever a key is written into a memory cell in the hashing step it is deactivated, and its bucket size decreases. The modified algorithm performs at least as well as the implementation in which a bucket is deactivated only if all of its keys are mapped injectively. The total memory used by the modified algorithm and the size of the representation of the hash table do not change.

Allocation step We first note that the algorithm can be carried out without using bucket representatives at all. Allocation of memory blocks is done using hash functions, as in Lemma 6.1; each processor can individually compute the index of its memory block by evaluating the function g_t. This function is selected by a designated processor and its representation ($6\lg m$ bits) is read in constant time by all processors.

We further modify the algorithm, so that the hashing step is carried out by *all* active buckets. That is, even buckets that collided in the allocation step will participate in the hashing step. This modification can only serve to improve the performance of the algorithm, since even while sharing a block with another bucket the probability that a bucket finds an injective function into that block is not zero. This modification eliminates the concurrent memory access needed for detecting failures in the allocation step.

Hashing step The selection of a level-2 hash function is done as in the hashing step described in Section 6. As can be seen from (39), only four global parameters should be selected and made available to all processors; this can be done in constant time.

It remains to eliminate the concurrent memory access required for determining if the level-2 function of any single bucket was injective. Whenever a key is successfully hashed by this function, it is deactivated even if other keys in the same bucket were not successfully hashed. Thus, keys of the same bucket may be stored in the hash table using different level-2 hash functions.

The two steps of an iteration in the hashing algorithm are summarized in Figure 3.

> Let x be an active key in a bucket $i = f(x)$. The processor assigned to x executes the following steps.
>
> *Allocation*: Compute $g_t(i)$, the index of the memory block selected to the bucket of x, where g_t is defined by (38).
>
> *Hashing*: Determine h_i, the level-2 hash function selected by the bucket of x, where h_i is defined by (39). Write x into cell $h_i(x)$ in memory block $g_t(i)$ and read the contents of that cell; if x was written then the key x becomes inactive.

Figure 3: *Implementation of iteration t in the hashing algorithm on* ROBUST

Lookup algorithm The search for a key x is done as follows. Let $i = f(x)$; for $t = 1, 2, \ldots$ read position $h_i(x)$ in the memory block $g_t(i)$ in the appropriate array. (All random bits that were used in the hash table construction algorithm are assumed to be recorded and available.) The search is terminated when either x is found, or else when t exceeds the number of iterations in the construction algorithm.

The lookup algorithm requires $O(\lg \lg n)$ iterations in the worst case. However, for any key $x \in S$ the expected lookup time (over all the random selections made by the hashing algorithm) is $O(1)$.

An alternative simplified implementation

Curiously, the sequence of modifications to the algorithm described in this section has lead to a *1-level* hashing scheme, i.e., to the elimination of indirect addressing. To see this, we observe that at iteration t an active key x is written into a memory cell ${g_t}^*(x)$, where the function ${g_t}^*(x)$ is dependent only on n and on the random selections made by the algorithm, but not on the input. An even simpler implementation of a 1-level hashing algorithm is delineated next.

At each iteration t, a new array T_t of size $3M_t$ is used, where M_t is as defined in (19). In addition, a function g_t as defined in (38) is selected at random. A processor representing an active key x in the iteration tries to write x into $T_t[g_t(x)]$, and then reads this cell. If x is successfully written in $T_t[g_t(x)]$ then x is deactivated. Otherwise, x remains active and the processor representing it carries on to the next iteration.

To see that the algorithm terminates in $O(\lg\lg n)$ iterations, we observe that the operation on keys in each iteration is the same as the operation on buckets in the allocation step of Section 6. Therefore, the analysis of Section 6 can be reused, substituting keys for buckets (and ignoring failures in the hashing step of the 2-level algorithm). The hash table consists of the collection of the arrays $T_1, T_2, \ldots$, and, as can be easily verified, is of linear size. A lookup query for a given key x is executed in $O(\lg\lg n)$ time by reading $T_t[g_t(x)]$ for $t = 1, 2, \ldots$.

8.3 Minimizing concurrent read requirements

The algorithms for construction of the hash table on TOLERANT and ROBUST can be modified to use concurrent-read from a single cell only. By allowing a pre-processing stage of $O(\lg n)$ time, concurrent read can be eliminated, implying that the ERCW model is sufficient. With these modifications, parallel lookups still require concurrent read, and their execution time increases to $O(\lg\lg n)$ in the worst case. Nevertheless, the expected time for lookup of any single key $x \in S$ is $O(1)$. The details are described next.

8.3.1 Concurrent read in the TOLERANT implementation

There are two types of concurrent read operations required by the modified algorithm. First, the sequence of $O(\lg\lg n)$ functions ${g_t}^*$ (or alternatively, g_t in the simplified implementation), must be agreed upon by all processors. Since each of these functions is represented by $O(\lg u)$ bits, its selection can be broadcasted at the beginning of the iteration through the concurrent-read cell.

The single cell concurrent read requirement for broadcasting can be eliminated by adding an $O(\lg n)$-time pre-processing step for the broadcasting. (This is just a special case of simulating CRCW PRAM by EREW PRAM.)

The other kind of concurrent-read operation occurs when processors read a memory cell to verify that their hashing into that cell has succeeded. This operation can be replaced by the following procedure. For each memory cell, there is a processor standing by. Whenever a pair $\langle x, j\rangle$ is written into a cell, the processor assigned to that cell sends an acknowledgement to processor j by writing into a memory cell j in a designated array.

The lookup algorithm requires concurrent-read capabilities. In this sense, the lookup operation is more demanding than the construction of the hash table. A similar phenomenon was observed by Karp, Luby and Meyer auf der Heide [36] in the context of simulating a random access machine on a distributed memory machine. The main challenge in the design of their (parallel-hashing based) simulation algorithm was the execution of the read step. Congestions during the execution of the write step were resolved by attempting to write in several locations and using the first for which the write succeeded. It is more difficult to resolve read congestions since the cells in which values were stored are already determined. Indeed, the read operation constitutes the main run-time bottleneck in their algorithm.

8.3.2 Concurrent Read in the Robust implementation

The simplified 1-level hashing algorithm for construction of the hash table on Robust is modified as follows. We eliminate the step in which a processor with key x reads the contents of the cell $T_t[g_t(x)]$ after trying to write to that cell. Instead, we use the acknowledgement technique described above: A processor j handling an active key x writes $\langle x, j\rangle$ into the cell $T_t[g_t(x)]$. The processor standing by cell $T_t[g_t(x)]$ into which $\langle x, j\rangle$ is written, sends an acknowledgement to processor j.

Note that this implementation introduces a new type of failures: due to the unpredictability of the concurrent write operation in Robust, an acknowledgement for a successful hash may not be received. Consider for example the following situation: Let j be a processor whose key x did not collide. Let i, i' be two processors with colliding keys y, y', i.e., $g_t(y) = g_t(y')$. These two processors concurrently write the pairs $\langle y, i\rangle$ and $\langle y', i'\rangle$ into the cell $T_t[g_t(y)]$. The result of this concurrent write is arbitrary. In particular, it can be the pair $\langle x', j\rangle$, which would cause the processor standing by the cell $T_t[g_t(y)]$ to garble the acknowledgement sent to processor j. (Recall that an acknowledgement to processor j is implemented by writing into a memory location associated with j.)

The number of the new failures described above can be at most half the number of colliding keys. It is easy to verify that the analysis remains valid, since the number of these new failures in no more than the number of "hashing failures" accounted for in Section 5.5, and which do not occur in this implementation.

9 Hashing of Multi-Sets

We conclude the technical discussion by briefly considering a variation of the hashing problem in which the input is a multi-set rather than a set. We first note that the analyses of exponential and doubly-exponential rate of decrease in the problem size is not affected by the possibility of multiple occurrences of the same key. This is a result of relying on estimates of the number of active *buckets* rather than the number of active keys. The number of distinct keys—not the number of keys—determines the probability of a bucket to find an injective function.

A predictable decrease in the number of active *keys* is essential for obtaining an optimal speedup algorithm. Unfortunately, the analysis in Section 7 with regard to the implementation of Section 5 does not hold. To understand the difficulty, consider the case where a substantial fraction of the input consists of copies of the same key. Then, with non-negligible probability this key may belong to a large bucket. The probability that this bucket deactivates in the first few iterations, in which the memory blocks are not sufficiently large, is too small to allow global decrease in the number of keys with high probability. Consequently, the rapid decrease in the number of buckets may not be accompanied by a similar decrease in the number of keys.

In contrast, the nature of the analysis in Section 4 makes it susceptible to an easy extension to multiple keys, which leads to an optimal speedup algorithm, albeit with expected performance only. Using the probabilistic induction lemma all that is required is to show that each copy of an active key stands a constant positive probability of deactivation at each iteration. Since the analysis is based on expectations only, there are no concerns regarding correlations between copies of the same key, or dependencies between different iterations. The details are left to the reader.

We also note that the model of computation required for a multi-set is COLLISION$^+$, since it must be possible to distinguish between the case of multiple copies of the same key being written into a memory cell, and the case where distinct keys are written. Also, the extensions of the hashing algorithms which only require concurrent read from a single memory cell can be used for hashing with multi-set input, but then a COLLISION$^+$ model, as opposed to ROBUST, must be assumed.

We finally observe that the hashing problem with a multi-set as input can be reduced into the ordinary hashing problem (in which the input consists of a set), by a procedure known as *leaders election*. This procedure selects a single representative from among all processors which share a value. By using an $O(\lg\lg n)$-time, linear-work leaders election algorithm which runs on TOLERANT [24] we have

Theorem 2 *Given a multi-set of n keys drawn from a universe U, the hashing problem can be solved using $O(n)$ space: (i) in $O(\lg\lg n)$ time with high probability, using n processors, or (ii) in $O(\lg\lg n \lg^* n)$ time and $O(n)$ operations with high probability. The algorithms run on* TOLERANT.

Conversely, note that any hashing algorithm, when run on ARBITRARY, solves the leaders election problem. In particular, the simple 1-level hashing algorithm for ROBUST, when implemented on ARBITRARY with a multi-set as input, gives a simple leaders election algorithm.

Consider now another variant of the multi-set hashing problem in which a data record is associated with each key. The natural semantics of this problem is that multiple copies of the same key can be inserted into the hash table only if their data records are identical. Processors representing copies of a key with conflicting data records should terminate the computation with an error code.

The COLLISION$^+$ model makes it easy enough to extend the implementations discussed above to accommodate this variant.

A more sophisticated semantics, in which the data records should be consolidated, requires a different treatment, e.g., by applying an integer sorting algorithm on the hashed keys (see [39]).

10 Conclusions

We presented a novel technique of hashing by oblivious execution. By using this technique, algorithms for constructing a perfect hash table which are fast, simple, and efficient, were made possible. The running time obtained is best possible in a model in which keys are only handled in their original processors.

The number of random bits consumed by the algorithm is $\Theta(\lg\lg u + \lg n \lg\lg n)$. An open question is to close the gap between this number and the $\Theta(\lg\lg u + \lg n)$ random bits that are consumed in the sequential hashing algorithm of [11].

The program executed by each processor is extremely simple. Indeed, the only coordination between processors is in computing the AND function, when testing for injectiveness. In the implementation on the ROBUST model, even this coordination is eliminated.

The large constants hidden under the "Oh" notation in the analysis may render the described implementations still far from being practical. We believe that the constants can be substantially improved without compromising the simplicity of the algorithm, by a more careful tuning of the parameters and by tightened analysis. This may be an interesting subject of a separate research.

The usefulness of the oblivious execution approach presented in this paper is not limited to the hashing problem alone. We have adopted it in [24] for simulations among sub-models of the CRCW PRAM. As in the hashing algorithm, keys are partitioned into subsets. However, this partition is arbitrary and given in the input, and for each subset the maximum key must be computed.

Subsequent work

The oblivious execution technique for hashing from Section 3 and its implementation from Section 4 were presented in preliminary form in [21]. Subsequently, our oblivious execution technique was used several times to obtain improvements in running time of parallel hashing algorithms: Matias and Vishkin [38] gave an $O(\lg^* n \lg\lg^* n)$ expected time algorithm; Gil, Matias, and Vishkin [26] gave a tighter failure probability analysis for the algorithm in [38], yielding $O(\lg^* n)$ time with high probability; similar improvement (from $O(\lg^* n \lg\lg^* n)$ expected time to $O(\lg^* n)$ time with high probability), was described independently by Bast and Hagerup [3].

An $O(\lg^* n)$ time hashing algorithm is used as a building block in a parallel dictionary algorithm presented in [26]. (A parallel dictionary algorithm supports in parallel batches of operations *insert*, *delete*, and *lookup*.) The oblivious execution technique has an important role in the implementation of insertions into the dictionary. The dictionary algorithm runs in $O(\lg^* n)$ time with high probability, improving the $O(n^\epsilon)$ time dictionary algorithm of Dietzfelbinger and Meyer auf der Heide [13]. The dictionary algorithm can be used to obtain a space efficient implementation of any parallel algorithm, at the cost of a slowdown of at most $O(\lg^* n)$ time with high probability.

The above hashing algorithms use the log-star paradigm of [38], relying extensively on processor reallocation, and are not as simple as the algorithm presented in this paper. Moreover, they require a substantially larger number of random bits.

Karp, Luby and Meyer auf der Heide [36] presented an efficient simulation of a PRAM on a distributed memory machine in the doubly-logarithmic time level, improving over previous simulations in the logarithmic time level. The use of a fast parallel hashing algorithm is essential in their result; the algorithm presented here is sufficient to obtain it.

Goldberg, Jerrum, Leighton and Rao [28] used techniques from this paper to obtain an $O(h + \lg\lg n)$ randomized algorithm for the h-relation problem on the optical communication parallel computer model.

Gibbons, Matias and Ramachandran [18] adapted the algorithm presented here to obtain a low-contention parallel hashing algorithm for the QRQW PRAM model [19]; this implies an efficient hashing algorithm on Valiant's BSP model, and hence on hypercube-type non-combining networks [44].

Acknowledgments *We thank Martin Dietzfelbinger and Faith E. Fich for providing helpful comments. We also wish to thank Uzi Vishkin and Avi Wigderson for early discussions. Part of this research was done during visits of the first author to AT&T Bell Laboratories, and of the second author to the University of British Columbia. We would like to thank these institutions for supporting these visits. Many valuable comments made by two anonymous referees are gratefully acknowledged.*

References

[1] N. Alon and J. H. Spencer. *The Probabilistic Method.* John Wiley & Sons, Inc., 1991.

[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] H. Bast and T. Hagerup. Fast and reliable parallel hashing. In *3rd ACM Symp. on Parallel Algorithms and Architectures*, pages 50–61, July 1991.

[4] P. Beame and J. Håstad. Optimal bounds for decision problems on the CRCW PRAM. *J. ACM*, 36(3):643–670, 1989.

[5] P. C. Bhatt, K. Diks, T. Hagerup, V. C. Prasad, T. Radzik, and S. Saxena. Improved deterministic parallel integer sorting. *Information and Computation*, 94:29–47, 1991.

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

[7] A. Borodin, J. E. Hopcroft, M. S. Paterson, W. L. Ruzzo, and M. Tompa. Observations concerning synchronous parallel models of computation. Manuscript, 1980.

[8] L. J. Carter and M. N. Wegman. Universal classes of hash functions. *J. Comput. Syst. Sci.*, 18:143–154, 1979.

[9] B. S. Chlebus, K. Diks, T. Hagerup, and T. Radzik. New simulations between CRCW PRAMs. In *Proc. 7th Int. Conference on Fundamentals of Computation Theory, Springer LNCS 380*, pages 95–104, 1989.

[10] B. Chor and O. Goldreich. On the power of two-point based sampling. *Journal of Complexity*, 5:96–106, 1989.

[11] M. Dietzfelbinger, J. Gil, Y. Matias, and N. Pippenger. Polynomial hash functions are reliable. In *Proc. 19th Int. Colloquium on Automata Languages and Programming, Springer LNCS 623*, pages 235–246, July 1992.

[12] M. Dietzfelbinger, A. R. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4), 1994.

[13] M. Dietzfelbinger and F. Meyer auf der Heide. An optimal parallel dictionary. In *1st ACM Symp. on Parallel Algorithms and Architectures*, pages 360–368, 1989.

[14] M. Dietzfelbinger and F. Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *Proc. 17th Int. Colloquium on Automata Languages and Programming, Springer LNCS 443*, pages 6–19, 1990.

[15] F. E. Fich, P. L. Ragde, and A. Wigderson. Relations between concurrent-write models of parallel computation. *SIAM Journal on Computing*, 17:606–627, 1988.

[16] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31(3):538–544, 1984.

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

[18] P. B. Gibbons, Y. Matias, and V. L. Ramachandran. Efficient low-contention parallel algorithms. In *6th ACM Symp. on Parallel Algorithms and Architectures*, pages 236–247, June 1994. Full version to appear in *JCSS*, Special issue for SPAA'94.

[19] P. B. Gibbons, Y. Matias, and V. L. Ramachandran. The QRQW PRAM: Accounting for contention in parallel algorithms. In *Proc. 5th ACM-SIAM Symp. on Discrete Algorithms*, pages 638–648, Jan. 1994.

[20] J. Gil. Fast load balancing on a PRAM. In *Proc. 3rd IEEE Symp. on Parallel and Distributed Computing*, pages 10–17, Dec. 1991.

[21] J. Gil and Y. Matias. Fast hashing on a PRAM—designing by expectation. In *Proc. 2nd ACM-SIAM Symp. on Discrete Algorithms*, pages 271–280, Jan. 1991.

[22] J. Gil and Y. Matias. Designing algorithms by expectations. *Inf. Process. Lett.*, 51(1):31–34, 1994.

[23] J. Gil and Y. Matias. An effective load balancing policy for geometric decaying algorithms. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, 1994. To appear in *Journal of Parallel and Distributed Computing.*

[24] J. Gil and Y. Matias. Fast and efficient simulations among CRCW PRAMs. *Journal of Parallel and Distributed Computing*, 23(2):135–148, 1994.

[25] J. Gil and Y. Matias. Simple fast parallel hashing. In *Proc. 21st Int. Colloquium on Automata Languages and Programming, Springer LNCS 820*, pages 239–250, July 1994.

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

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

[28] L. A. Goldberg, M. Jerrum, F. T. Leighton, and S. B. Rao. Doubly logarithmic communication algorithms for optical communication parallel computers. In *5th ACM Symp. on Parallel Algorithms and Architectures*, pages 300–309, 1993.

[29] L. M. Goldschlager. A universal interconnection pattern for parallel computers. *J. ACM*, 29(4):1073–1086, 1982.

[30] M. T. Goodrich, Y. Matias, and U. Vishkin. Optimal parallel approximation algorithms for prefix sums and integer sorting. In *Proc. 5th ACM-SIAM Symp. on Discrete Algorithms*, pages 241–250, Jan. 1994.

[31] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics.* A Foundation for Computer Science. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, May 1989.

[32] V. Grolmusz and P. L. Ragde. Incomparability in parallel computation. *Discrete Applied Mathematics*, 29:63–78, 1990.

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

[34] T. Hagerup and T. Radzik. Every robust CRCW PRAM can efficiently simulate a Priority PRAM. In *2nd ACM Symp. on Parallel Algorithms and Architectures*, pages 117–124, 1990.

[35] J. JáJá. *Introduction to Parallel Algorithms.* Addison-Wesley Publishing Company, Inc., 1992.

[36] R. M. Karp, M. Luby, and F. Meyer auf der Heide. Efficient PRAM simulation on a distributed memory machine. In *Proc. 24th ACM Symp. on Theory of Computing*, pages 318–326, 1992.

[37] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1973.

[38] Y. Matias and U. Vishkin. Converting high probability into nearly-constant time—with applications to parallel hashing. In *Proc. 23rd ACM Symp. on Theory of Computing*, pages 307–316, May 1991.

[39] Y. Matias and U. Vishkin. On parallel hashing and integer sorting. *Journal of Algorithms*, 12(4):573–606, 1991.

[40] K. Mehlhorn. *Data Structures and Algorithms I: Sorting and Searching.* Springer-Verlag, Berlin Heidelberg, 1984. EATCS Monographs on Theoretical Computer Science.

[41] K. Mehlhorn and A. Tsakalidis. Data structures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, pages 301–341. North-Holland, Amsterdam, 1990.

[42] Y. Shiloach and U. Vishkin. An $O(\lg n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3:57–67, 1982.

[43] A. Siegel. On universal classes of fast high performance hash functions, their time-space tradeoff, and their applications. In *Proc. 30th IEEE Symp. on Foundations of Computer Science*, pages 20–25, 1989.

[44] L. G. Valiant. General purpose parallel architectures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, chapter 18, pages 944–971. Elsevier Science Publishers B.V., 1990.

A Analysis of Moments of the Bucket Distribution

The material of this appendix is taken from [11], and is given here for completeness.

The reader is reminded that we assume the probability space where $h : U \to [0, m-1]$ is selected at random from a suitable class of hash functions. Also, the set $S \subseteq U$, $|S| = n$, is fixed, and m/n is denoted by μ.

We use the following notation: for a sequence $\mathbf{x}$, $\overline{\mathbf{x}}$ is the set of elements comprising $\mathbf{x}$.

For $\mathbf{x} \in U^r$, $\mathbf{x} = (x_1, \ldots, x_r)$, let

$$\delta(\mathbf{x}) = \begin{cases} 1 & \text{if } h(x_1) = \cdots = h(x_r) \\ 0 & \text{otherwise} \end{cases} \quad .$$

If h is r-wise independent[3] then

$$\mathbf{E}\left(\delta(\mathbf{x})\right) = \frac{1}{m^{j-1}} \ , \tag{42}$$

where $j = |\overline{\mathbf{x}}|$. Polynomial hash functions of degree $r-1$ are "almost" r-wise independent as they satisfy a weaker condition:

$$\mathbf{E}\left(\delta(\mathbf{x})\right) \le \left(\frac{2}{m}\right)^{j-1} \quad . \tag{43}$$

With the above notation we can write

$$B_r = \sum_{0 \le i < m} \binom{s_i}{r} = \frac{1}{r!} \sum_{\substack{\mathbf{x} \in S^r \\ |\overline{\mathbf{x}}| = r}} \delta(\mathbf{x})$$

and therefore for degree-$(k-1)$ polynomial hash functions

$$\begin{aligned} \mathbf{E}\left(B_r\right) &= \frac{1}{r!} \sum_{\substack{\mathbf{x} \in S^r \\ |\overline{\mathbf{x}}| = r}} \mathbf{E}\left(\delta(\mathbf{x})\right) \\ &\overset{\text{by (43)}}{\le} \binom{n}{r} \left(\frac{2}{m}\right)^{r-1} \\ &\le m \frac{2^{r-1}}{r!} \mu^{-r} \quad . \end{aligned}$$

[3]A function is r-wise independent if it assumes fully random values on all sets of up to r keys. For example, the class of all linear transformation of degree r from one vector space over a field to another vector space over a field is r-wise independent.

In particular, for linear hash functions Fact 2.2 follows. For r-wise independent functions (42) can be used to derive

$$\mathbf{E}\left(B_r\right) = \binom{n}{r} m^{1-r} \leq \frac{m}{r!\mu^r} \ . \tag{44}$$

Also,

$$A_r = \sum_{0\leq i<m} s_i^r = \sum_{\mathbf{x}\in S^r} \delta(\mathbf{x}) = \sum_{1\leq j<r} \sum_{\substack{\mathbf{x}\in S^r\\ |\overline{\mathbf{x}}|=j}} \delta(\mathbf{x}) \ . \tag{45}$$

Therefore

$$\mathbf{E}\left(A_r\right) = \sum_{1\leq j<r} \sum_{\substack{\mathbf{x}\in S^r\\ |\overline{\mathbf{x}}|=j}} \mathbf{E}\left(\delta(\mathbf{x})\right) \ .$$

The number of different sequences $\mathbf{x} \in S^r$ such that $|\overline{\mathbf{x}}| = j$ is

$$|\{\mathbf{x} \in S^r : |\overline{\mathbf{x}}| = j\}| = \frac{n!}{(n-j)!} \left\{ {r \atop j} \right\} \leq n^j \left\{ {r \atop j} \right\} \ .$$

For degree-$(k-1)$ polynomial hash functions we use (43) to obtain

$$\mathbf{E}\left(A_r\right) \leq \sum_{1\leq j\leq r} n^j \left\{ {r \atop j} \right\} \left(\frac{2}{m}\right)^{j-1} = n \sum_{1\leq j\leq r} \left\{ {r \atop j} \right\} \left(\frac{2}{\mu}\right)^{j-1} \tag{46}$$

thereby proving Fact 2.6. For r-wise independent hash functions we use (42) to obtain

$$\mathbf{E}\left(A_r\right) = \sum_{1\leq j\leq r} \frac{n!}{(n-j)!} \left\{ {r \atop j} \right\} \frac{1}{m^{j-1}} \leq \sum_{1\leq j\leq r} \left\{ {r \atop j} \right\} j!\mathbf{E}\left(B_j\right) \ . \tag{47}$$

Using the inversion formula for Stirling numbers (e.g. [31, page 250]) we get

$$\mathbf{E}\left(B_r\right) = \frac{1}{r!} \sum_{1\leq j\leq r} (-1)^{r-j} \left[{r \atop j} \right] \mathbf{E}\left(A_j\right) \ , \tag{48}$$

where $\left[{r \atop j} \right]$ is the Stirling number of the first kind.

We now turn to computing the variance of A_r for $2r$-wise independent hash functions. By Equation 45

$$\begin{aligned}
\mathbf{Var}\left(A_r\right) &= \sum_{\mathbf{x}\in S^r} \mathbf{Var}\left(\delta(\mathbf{x})\right) + \sum_{\mathbf{x},\mathbf{y}\in S^r} \mathbf{CoVar}\left(\delta(\mathbf{x}), \delta(\mathbf{y})\right) \\
&\leq \sum_{\mathbf{x}\in S^r} \mathbf{E}\left(\delta(\mathbf{x})\right) + \sum_{\substack{\mathbf{x},\mathbf{y}\in S^r\\ \overline{\mathbf{x}}\cap\overline{\mathbf{y}}=\emptyset}} \mathbf{CoVar}\left(\delta(\mathbf{x}), \delta(\mathbf{y})\right) + \sum_{\substack{\mathbf{x},\mathbf{y}\in S^r\\ \overline{\mathbf{x}}\cap\overline{\mathbf{y}}\neq\emptyset}} \mathbf{CoVar}\left(\delta(\mathbf{x}), \delta(\mathbf{y})\right) \\
&\leq \mathbf{E}\left(A_r\right) + \sum_{\substack{\mathbf{x},\mathbf{y}\in S^r\\ \overline{\mathbf{x}}\cap\overline{\mathbf{y}}\neq\emptyset}} \mathbf{CoVar}\left(\delta(\mathbf{x}), \delta(\mathbf{y})\right) \ ,
\end{aligned}$$

since all $\delta(\mathbf{x})$ receive value from $\{0,1\}$. If the sets $\overline{\mathbf{x}}$ and $\overline{\mathbf{y}}$ are not disjoint then the co-variance of $\delta(\mathbf{x})$ and $\delta(\mathbf{y})$ is dependent only on the parameters

$$\begin{array}{rcl} j_0 & = & |\overline{\mathbf{x}} \cap \overline{\mathbf{y}}| \\ j_0 + j_1 & = & |\overline{\mathbf{x}}| \\ j_0 + j_2 & = & |\overline{\mathbf{y}}| \end{array} \quad , \tag{49}$$

and is given by

$$\mathbf{CoVar}(\delta(\mathbf{x}), \delta(\mathbf{y})) = m^{1-(j_0+j_1+j_2)} - m^{1-(j_0+j_1)} m^{1-(j_0+j_2)} \leq m^{1-(j_0+j_1+j_2)} \quad . \tag{50}$$

For given j_0, j_1, j_2, the number of distinct settings of $\mathbf{x}, \mathbf{y} \in S$ which satisfy (49) is

$$\binom{n}{n-(j_0+j_1+j_2)} \left\{ {r \atop j_0+j_1} \right\} \left\{ {r \atop j_0+j_2} \right\} \leq \frac{n^{j_0+j_1+j_2}}{(j_0+j_1+j_2)!} \left\{ {r \atop j_0+j_1} \right\} \left\{ {r \atop j_0+j_2} \right\} \ .$$

Using the above and (50) we obtain

$$\begin{array}{rcl} \mathbf{Var}(A_r) - \mathbf{E}(A_r) & \leq & \displaystyle\sum_{\substack{j_1, j_2 \leq r \\ 1 \leq j_0 \leq 2r-(j_1+j_2)}} \frac{n^{j_0+j_1+j_2}}{(j_0+j_1+j_2)!} \left\{ {r \atop j_0+j_1} \right\} \left\{ {r \atop j_0+j_2} \right\} m^{1-(j_0+j_1+j_2)} \\ & \leq & m \displaystyle\sum_{\substack{j_1, j_2 \leq r \\ 1 \leq j_0 \leq 2r-(j_1+j_2)}} \frac{1}{(j_0+j_1+j_2)! \mu^{j_0+j_1+j_2}} \left\{ {r \atop j_0+j_1} \right\} \left\{ {r \atop j_1+j_2} \right\} \ . \end{array}$$

The number of terms in the above summation is not dependent on n or m. If $m = O(n)$ then for any fixed k

$$\mathbf{Var}(A_r) = O(n)$$

which (with Chebyshev's inequality) proves Fact 2.7 for $2r$-wise independent hash functions. The extension for polynomial hash functions is more involved and is not given here.