Algorithmica (1996) 16: 517–542

Algorithmica

Efficient PRAM Simulation on a Distributed Memory Machine

R. M. Karp,[1] M. Luby,[2] and F. Meyer auf der Heide[3]

Abstract. We present algorithms for the randomized simulation of a shared memory machine (PRAM) on a Distributed Memory Machine (DMM). In a PRAM, memory conflicts occur only through concurrent access to the same cell, whereas the memory of a DMM is divided into modules, one for each processor, and concurrent accesses to the same module create a conflict. The *delay* of a simulation is the time needed to simulate a parallel memory access of the PRAM. Any general simulation of an m processor PRAM on an n processor DMM will necessarily have delay at least m/n. A randomized simulation is called *time-processor optimal* if the delay is $O(m/n)$ with high probability. Using a novel simulation scheme based on hashing we obtain a time-processor optimal simulation with delay $O(\log\log(n)\log^*(n))$. The best previous simulations use a simpler scheme based on hashing and have much larger delay: $\Theta(\log(n)/\log\log(n))$ for the simulation of an n processor PRAM on an n processor DMM, and $\Theta(\log(n))$ in the case where the simulation is time-processor optimal.

Our simulations use several (two or three) hash functions to distribute the shared memory among the memory modules of the PRAM. The stochastic processes modeling the behavior of our algorithms and their analyses based on powerful classes of universal hash functions may be of independent interest.

Key Words. PRAM, Simulation, Distributed memory machine, Universal hashing, Stochastic modeling, Balls and bins.

1. Introduction.

Parallel machines that communicate via a shared memory (parallel random access machines, PRAMs) are the most commonly used machine model for describing parallel algorithms. The PRAM is relatively comfortable to program, because the programmer does not have to allocate storage within a distributed memory or specify interprocessor communication. On the other hand, shared memory machines are very unrealistic from the technological point of view, because, on large machines, a parallel shared memory access can only be realized at the cost of a significant time delay. A more realistic model is the distributed memory machine (DMM), in which the memory is divided into a limited number of memory modules, one module per processor. Each

[1] Computer Science Department, University of Washington, USA. Research partially supported by NSF/DARPA Grant CCR-9005448. Work was done while at the University of California at Berkeley and the International Computer Science Institute, Berkeley, CA.
[2] International Computer Science Institute, Berkeley, CA, 94704, USA, and University of California, Berkeley, CA 94720, USA. Research partially supported by National Science Foundation Operating Grant CCR-9016468, National Science Foundation Operating Grant CCR-9304722, United States–Israel Binational Science Foundation Grant No. 89-00312, United States–Israel Binational Science Foundation Grant No. 92-00226, and ESPRIT BR Grant EC-US 030.
[3] Heinz Nixdorf Institute and Computer Science Department, University of Paderborn, D-33095 Paderborn, Germany. Part of work was done during a visit at the International Computer Science Institute at Berkeley; supported in part by DFG-Forschergruppe "Effiziente Nutzung massiv paralleler Systeme, Teilprojekt 4," and by the Esprit Basic Research Action Nr. 7141 (ALCOM II).

Received August 12, 1993; revised November 10, 1994. Communicated by M. Luby.

module can respond to only one access request at a time. Thus DMMs exhibit the phenomenon of *memory contention*, in which an access request is delayed because of a concurrent request to the same module.

In an effort to understand the effects of memory contention on the performance of parallel computers, several authors have investigated the simulation of shared-memory machines on DMMs. In this paper we present substantial improvements over the most efficient simulations previously known.

The paper is organized as follows. In the next two subsections we summarize previous work and the new results presented in this paper. Section 2 contains more detailed descriptions of the computation models. Section 4 describes the universal classes of hash functions we require and discusses stochastic processes that capture the essential features of our simulations. The remaining sections describe several variants of our shared memory simulations. The reader wishing to gain a quick insight into our algorithms without detailed probabilistic analysis may wish to skip Sections 5 and 6 in a first pass through the paper.

1.1. *Previous Work.* Let p denote the size of the shared memory of a PRAM, and let n be the number of processors and memory modules of a DMM.

Each of the previous randomized algorithms for simulating a PRAM on a DMM uses a single hash function $h\colon [p] \to [n]$ randomly chosen from a *universal* class of hash functions, to distribute the shared memory cells (we say "keys" for short) among the memory modules of the DMM.[4] Informally, a class of hash function is universal, if a random member of it has statistical properties not too far away from a random function, but can be stored on little space and can be evaluated fast. The notion of universality was first introduced in [4].

Some of the simulations assume a complete interconnection network between the processors and the memory modules; others assume a sparse interconnection network such as a butterfly or a hypercube.

The *delay* of a simulation is the time needed to simulate a parallel memory access of the PRAM. For those simulations that are based on a single hash function the delay is governed by the following quantities:

- The *hash evaluation time*, i.e., the time to evaluate h.
- The *memory contention*, i.e., the maximum number of shared memory accesses executed in one PRAM step which are mapped to the same module under h.
- In the case of a sparse interconnection network, the *routing time*, i.e., the time needed to route read and write requests from processors to memory modules, and to transmit the results of read requests back to the requesting processors.

In [15] and [20] it is shown that, on a butterfly network, expected routing time $O(\log(n))$ can be achieved, which clearly is asymptotically optimal. The expected contention can be made as small as $O(\log(n)/\log\log(n))$, if $\log(n)$-universal hash-functions as introduced in [4] are used. These hash functions have evaluation time $O(\log(n))$. Thus these simulations have delay $O(\log(n))$.

[4] In this paper $[n]$ denotes $\{1, 2, \ldots, n\}$.

If a complete interconnection network is assumed then the delay can be reduced. In [19], $\log(n)/\log\log(n)$-universal hash functions from [4] are used, yielding delay $O(\log(n)/\log\log(n))$. For any scheme that uses a single hash function to distribute the keys among n memory modules, the expected contention is necessarily $\Omega(\log(n)/\log\log(n))$. It is easily shown that this holds if the hash function behaves like a random function, and that no class of hash function can yield lower expected contention. Thus any improvement must stem from the use of more than one hash function.

By introducing parallel slackness—i.e., simulating a PRAM on a DMM with fewer processors—we can obtain *time-processor optimal simulations*, in which the expected delay is proportional to the ratio between m, the number of processors in the PRAM, and n, the number of processors in the DMM. Time-processor optimality can only be achieved on completely connected DMMs, and only if the hash functions used have constant evaluation time. In such simulations each processor of the DMM simulates m/n processors of the PRAM. Thus each simulation step has to satisfy m/n memory access requests from each processor. In devising time-processor optimal simulations the object is to minimize the delay or, equivalently, to minimize m as a function of n. It is easily seen that, in any time-processor optimal simulation that uses a single hash function to distribute the keys, the expected memory contention, and hence the expected delay, must be $\Omega(\log(n))$.

The first time-processor optimal simulation was published in [16]. It simulates an $n^{1+\varepsilon}$ processor PRAM on an n processor DMM with optimal expected delay $O(n^{\varepsilon})$, for arbitrary $\varepsilon > 0$. In [23] a time-processor optimal simulation of EREW PRAMs on DMMs is presented with expected delay $O(\log(n))$, using hash functions introduced in [21]. In [8] the same result is shown for CRCW PRAMs, using a new class of hash functions.

1.2. *New Results.* In this paper we have chosen to assume a complete interconnection network in order to avoid confounding the effects of memory contention with the effects of routing delays, and to make possible the construction of time-processor optimal simulations. Our main result is a time-processor optimal randomized simulation of an EREW PRAM with delay $O(\log\log(n)\log^*(n))$ and a simulation of a CRCW PRAM with the same delay, where the time-processor product is only away from optimal by a factor $\log^*(n)$. The delay bound is very reliable; it is guaranteed *with high probability, i.e., with probability exceeding* $1 - O(n^{-\ell})$ for arbitrary $\ell > 0$. (Hereafter, this is what we mean whenever we use the term "with high probability.")

The simulation uses a novel scheme which is more involved than the simple hashing scheme used in the previous results. We show how to speed up the simulation of a read step of an n processor PRAM on an n processor DMM by using two or more hash functions, and thus making the contents of each PRAM cell accessible in two or more places. We speed up the simulation of a write step by allowing delayed executions of write instructions: whenever memory contention prevents a write request from being executed during the present memory cycle, the request is stored in a parallel hash table. We show that with high probability the size of this table of deferred write requests never exceeds $O(n)$. Thus, we can distribute the table among the modules so that accesses to it can be performed in constant time.

The analysis of our simulation depends on the properties of a particular $\sqrt{n}$-universal class of hash functions which combines the constructions given in [8] and [21]. The structure of these hash functions enables us to analyze the delay in our simulation using a powerful martingale tail estimate that was derived independently in [2] and [18].

2. Computation Models.

A PRAM consists of processors $P_1, \dots, P_m$ and a shared memory with cells $U = [p]$. The processors work synchronously and have random access to the shared memory cells, each of which can store an integer. We consider EREW PRAMs where concurrent access to the same shared memory cell is forbidden, as well as CRCW PRAMs where such an access is allowed. In the latter case we assume the ARBITRARY write conflict resolution: If several processors want to write to the same cell simultaneously, an arbitrary one of them succeeds. The computation of the PRAM has to be correct no matter which one succeeds.

A DMM has processors $Q_1, \dots, Q_n$ which communicate via a distributed memory consisting of n memory modules $M_1, \dots, M_n$. Each module has a *communication window*. A module can read from or write into its window. From the viewpoint of the processors, a window acts like a shared memory cell of a CRCW PRAM with ARBITRARY write conflict resolution.

Our simulations of PRAMs on DMMs use known algorithms (for, e.g., perfect hashing, see Section 4), originally described for PRAMs. All these algorithms use linear space only. The next lemma states that such algorithms can be executed also on a DMM with constant delay.

LEMMA 2.1. *An n-processor CRCW PRAM with m shared memory cells can be simulated on an n-processor DMM with worst case delay $O(m/n)$.*

PROOF. Partition the shared memory of the PRAM, i.e. each module gets at most $\lceil m/n \rceil$ shared memory cells. Now any parallel access to the shared memory can be simulated in time $O(m/n)$ by the DMM. □

Our simulations need the capability of the DMM to execute concurrent reads from and concurrents writes into the communication windows. On the other hand the specific rule how to resolve write conflicts is not of major importance, because several authors have shown efficient simulations among parallel machines with different write conflict resolutions [5], [6], [10], [11], [13], [14], [22]. These simulations are described for PRAMs, but can be transfered to DMMs using Lemma 2.1.

For example the TOLERANT rule suffices: If several processors want to write to the same communication window, then its contents remains unchanged. This can be done without increase in the time-processor product, and with a $O(\log^*(n))$ increase for the delay, with high probability.

3. Outline of the Simulations.

Our PRAM simulations are based on the use of hash functions to distribute data among the memory modules of a DMM. The properties of the hash functions are complex, as are the analyses of the basic stochastic processes underlying the simulations. However, the design of the simulations was guided by a clear intuition based on the idealized assumption that the hash functions are completely independent random functions; i.e., that all hash functions have domain U and range $[n]$, that the value of hash function h at point x is determined by rolling a fair n-sided die, and that the values of distinct hash functions or the values of the same hash function at distinct points are completely independent.

All previously published shared memory simulations based on universal hashing use one hash function h to distribute the shared memory among the modules of the DMM. As already noted in the introduction, such simulations inherently exhibit the phenomenon of memory contention: If h is randomly drawn from an arbitrary universal class of hash functions, then the expected contention is $\Omega(\log(n)/\log\log(n))$.

In order to get faster simulations, we allow the use of more than one hash function. Thus each shared memory cell has several *copies*, maintained in several modules.

Assume that we use two hash functions h_1, h_2. Further assume that the correct values (with respect to the simulated PRAM) of each shared memory cell x are stored in the copies of x in $M_{h_1(x)}$ and $M_{h_2(x)}$. Then, if processor P_j wants to read cell x_i, it suffices to access only one arbitrary of the two copies of x_i, for $i \in [n]$.

We first note that in this situation, no inherent contention bound limits the performance of the memory access.

Consider the bipartite graph G with node sets $\{x_1, \ldots, x_n\}$ and $\{M_1, \ldots, M_n\}$, where edges go from x_i to $M_{h_1(x_i)}$ and $M_{h_2(x_i)}$, $i \in [n]$. As we assume that h_1 and h_2 are random and independent, it is easily checked that, with high probability, there is a constant number of matchings in G such that each x_i is incident to an edge in one of the matchings.

If now only the copy of x_i indicated by the edge of a matching is accessed, each module only receives a constant number of requests, i.e., the contention is constant.

On the other hand, the computation of the matchings is complicated. We present a simple algorithm, READ_2 from Section 7.2, which implicitly finds these matchings. READ_1 from Section 7.1 describes a simulation of a read step for the case that not two, but $\log\log(n)$ hash functions are used. The behaviors of the algorithms are modeled by the stochastic processes Process_1 and Process_3 in Section 6. Their analyses show that both simulations of a read step need time $O(\log\log(n))$, with high probability.

As we assume that all copies of all shared memory cells are always up to date, we have to make sure that simulations of a write step—P_i writes z_i to x_i, $i \in [n]$—update *all* copies of x_i with z_i, $i \in [n]$.

For this purpose we present a new idea—delayed writing—to speed up the simulation of a write step. Consider one hash function h. P_i wants to write to cell x_i, $i \in [n]$, i.e., wants to update the copy of x_i in $M_{h(x_i)}$. As memory contention prevents us from directly sending all updates to the respective modules, we first insert them into an intermediate data structure SM, which holds all not yet satisfied update requests. Then all update requests from SM are tried to be passed to the respective modules, but each module only accepts a constant number of requests.

This simulation of a write step is presented in Section 7. Process_2 in Section 6 models the shrinking and growing of the set of unsatisfied requests stored in SM. We show that this set will always be of linear size, with high probability.

For implementing SM we use a fast perfect hash table, based on results from [17] and [2], described in Section 4. This guarantees that time $O(\log^*(n))$ is sufficient to insert the new write requests into SM, with high probability, and that a parallel read in SM can be done in constant time.

Thus, with respect to one hash function, a write step can be simulated in time $O(\log^*(n))$, with high probability.

We finally present methods to make our simulations time-processor optimal.

4. Perfect Hashing and Approximate Compaction.

Let U be the set of addresses of the shared memory cells of the PRAM. We refer to a cell x as a key. The current contents of x is denoted $c(x)$. Our simulations maintain a small set of keys x, together with their current contents $c(x)$, in an intermediate data structure which has to be built efficiently on the DMM, and in which an efficient lookup can be performed. A lookup for a key x returns $c(x)$, if $(x, c(x))$ is stored in the data structure, and returns "failure" if key x is not in the data structure.

Let d be a positive constant. Define a *parallel hash table* SM with degree of parallelism n and set of addresses U as an array which contains a set of keys $S \subseteq U$, where $|S| \leq \gamma n$ for some $\gamma > 0$, together with a value $c(x)$ associated with a each key x. SM is evenly distributed among the modules. It supports the following operations:

BUILD$(S_1, S_2, \ldots, S_n)$

Input: A family of n not necessarily disjoint sets of key-value pairs $S_1, S_2, \ldots, S_n$, where, for all i, $|S_i| \leq \log^*(n)$, S_i is stored in M_i.

Output: If $\sum_i |S_i| \leq \gamma n$, then the operation produces a parallel hash table SM storing $\bigcup_i S_i$ and returns the value "success"; otherwise, it returns the value "failure." (Note: If x is in several sets S_i, it finally is only presented once in the hash table.)

LOOKUP(SM, X)

Input: A parallel hash table SM containing the set S, and an array of keys $X = (x_1, x_2, \ldots, x_n)$, x_i is stored in M_i.

Output: An array $Y = (y_1, y_2, \ldots, y_n)$ where: if $x_i \in S$ then y_i is the ordered pair $(c(x_i)$, "success$'')$; otherwise, y_i is equal to "failure." y_i is posted in the communication window of M_i.

HASH(SM, X)

Input: A parallel hash table SM containing the set S and an array X of key-value pairs $(x_i, c(x_i))$, stored in M_i.

Output: If $|S \cup X| \leq \gamma n$, then this operation sets SM equal to a parallel hash table storing $S \cup X$; otherwise, the operation returns "failure."

The papers [2] and [17] give randomized algorithms for realizing a parallel hash table for γn key-value pairs on an n processor PRAM. The inputs and outputs of the operations, as well as the parallel hash table itself, reside in the shared memory of the PRAM. The space required for the parallel hash table is $O(n)$. The LOOKUP operation runs in time $O(1)$ and, the BUILD and HASH operations run in time $O(\log^*(n))$ and perform $O(n)$ operations, with high probability. Note that the operation HASH is a little more complex than what is done in [2] or [17], because X is hashed in a nonempty hash table. For our version of HASH, we first collect the keys S stored in the hash table, and then hash $S \cup X$ to it. This is possible because $|S \cup X| = O(n)$. For our Simulation_2, a simpler implementation from [12] with delay $O(\log\log(n))$ would also suffice.

As shown in Lemma 2.1, these algorithms can be simulated on a DMM with constant slowdown.

5. Universal Families of Hash Functions.

Our simulations require us to distribute the shared memory of the PRAM among the memory modules of the DMM. We now discuss the hash functions that will be used for this purpose.

Let $U = [p]$, where $p > n$. For a function $h: U \to [n]$ and a set $S \subseteq U$ let $B_i^h = h^{-1}(i) \cap S$ be the ith *bucket of S under h*. The function h splits S into buckets $B_1^h, \dots, B_n^h$.

DEFINITION 5.1 (d-perfect). We call h *d-perfect on S*, if each B_i^h, $i = 1, \dots, n$, has size at most d.

Let $\mathcal{H}_{p,n}$ be a family of hash functions mapping U into $[n]$. In [4] the notion of universality for families of hash functions was introduced as a measure of the quality of the family for classical hashing purposes.

DEFINITION 5.2 (Universal Hashing). The family $\mathcal{H}_{p,n}$ is (μ, k)-universal, if for each $x_1 < \cdots < x_j \in U, y_1, \dots, y_j \in [n]$, $j \le k$, it holds that, if the hash function h is drawn with uniform probability from $\mathcal{H}_{p,n}$, then

$$\mathrm{Prob}[h(x_1) = y_1, \dots, h(x_j) = y_j] \le \frac{\mu}{n^j}.$$

Let p be prime, $p > n$. As building blocks for our hash functions we apply two types of universal classes. The first one is the class $\mathcal{H}_{p,n}^d \subseteq \{h\colon [p] \to [n]\}$ of functions $h(x) \bmod n$ where h is a polynomial of degree $d - 1$ over $\mathbb{Z}_p$. $\mathcal{H}_{p,n}^d$ was introduced by Carter and Wegman in [4]. It is a $(2, d)$-universal class. The second class $\bar{\mathcal{H}}_{n^k,n} \subseteq \{h\colon [n^k] \to [n]\}$ introduced by Siegel in [21] consists of more complicated functions. It is the first class with high degree of universality whose functions can be generated fast using little space and have constant evaluation time, if the universe is of size n^k for constant k. The following lemma lists important known properties of these classes.

LEMMA 5.1 (Properties of $\mathcal{H}_{p,n}^d$ and $\bar{\mathcal{H}}_{n^k,n}$). *Let d and k be constants independent of n.*

(a) *A random $h \in \mathcal{H}_{p,n}^d$ can be generated by a randomized sequential processor in constant time. A random $\bar{\mathcal{H}}_{n^k,n}$ can be generated by a randomized* DMM *with $\sqrt{n}$ processors in constant time.*
(b) *$h \in \mathcal{H}_{p,n}^d$ or $\bar{\mathcal{H}}_{n^k,n}$ can be evaluated in (sequential) constant time.*
(c) *$\mathcal{H}_{p,n}^d$ is $(2, d)$-universal.*
(d) *$\bar{\mathcal{H}}_{n^k,n}$ is $(1, \sqrt{n})$-universal for sufficiently large k.*

Let $\ell \geq 1$ be arbitrary, and let d and k be large enough relative to ℓ. Let $S \subseteq U$, $n \leq |S| \leq n^{11/10}$.

(e) *If h is randomly drawn from $\mathcal{H}^d_{p,\sqrt{n}}$, then* Prob[*$h$ is $2\,|S|/\sqrt{n}$-perfect on S*] $\geq 1 - n^{-\ell}$.

(f) *If h is randomly drawn from $\mathcal{H}^1_{p,n^k}$, then* Prob[*h is 1-perfect on S*] $\geq 1 - n^{-\ell}$.

(g) *Let $S' \subseteq U$, $|S'| \leq 2n^{3/4}$. If h is randomly drawn from $\mathcal{H}^d_{p,n}$ or $\bar{\mathcal{H}}_{n^k,n}$ then* Prob[*h is d-perfect on S'*] $\geq 1 - n^{-\ell}$.

Proof. The results (a) and (b) are obvious from the definition of the classes, (c) and (d) can be found in [4] for $\mathcal{H}^d_{p,n}$ and in [21] for $\bar{\mathcal{H}}_{n^k,n}$ and (e) is shown in [16]. The results (f) and (g) are shown in [7] for (μ, d)-universal classes; thus it applies to both of our classes because of (c) and (d). □

In [8] and [9] a new class of hash functions is introduced. We only present a special case sufficient for our considerations.

Definition 5.3 ($\mathcal{R}^d_{p,n}$). A particular function $h \in \mathcal{R}^d_{p,n}$ is specified by:

- A primary hash function $f \in \mathcal{H}^d_{p,\sqrt{n}}$.
- A secondary hash function $g \in \mathcal{H}^d_{p,n}$.
- A collection of offsets $\mathbf{a} = \langle a_1, \ldots, a_{\sqrt{n}} \rangle$, where $a_i \in [n]$.

The function h is defined in terms of $(f, g, \mathbf{a})$ as

$$h(x) = (g(x) + a_{f(x)}) \mod n.$$

In other words, to compute $h(x)$ we determine a base address $g(x)$ and add to it an offset determined by $f(x)$. Note that $h \in \mathcal{R}^d_{p,n}$ can be evaluated in time $O(d)$, i.e., in constant time if d is a constant. Choosing a random $h \in \mathcal{R}^d_{p,n}$ means choosing the parameters of the corresponding $f, g, \mathbf{a}$ independently at random. Note that a random h can be chosen by a randomized DMM with $\sqrt{n}$ processors in constant time, such that each processor knows f and g, and a_i is stored in M_i, $i = 1, \ldots, \sqrt{n}$. In this way, each processor can evaluate h on x in constant time, by reading $a_{f(x)}$ from $M_{f(x)}$.

For $\mathcal{R}^d_{p,n}$, [9] shows that for any given $S \subseteq U$, $|S| \leq n^{11/10}$, a randomly chosen and fixed (f, g) pair will have, with high probability, distributional properties with respect to how S is mapped by random $\mathbf{a}$ that are very similar to the properties that hold if completely random functions are used to map S.

Definition 5.4 (d-goodness). Let $f\colon U \to [\sqrt{n}]$ and $g\colon U \to [n]$ be hash functions. Let d be an integer and let $S \subseteq U$. (f, g) is d-good for S if f is $2\,|S|/\sqrt{n}$-perfect on S (i.e., each bucket of S under f has size at most twice the average bucket size), and g is d-perfect on each bucket of S under f.

The following lemma is implicitly used in [9]. It follows directly from Lemma 5.1(e) and (f).

LEMMA 5.2. *Let $S \subseteq U$, $n \le |S| \le n^{11/10}$. In the context of $\mathcal{R}^d_{p,n}$, for each $\ell \ge 0$ there is $d \ge 1$ such that a random pair (f, g) is d-good for S with probability $1 - n^{-\ell}$.*

For our time-processor optimal simulations we need a class of hash functions with constant evaluation time which has the same two-level structure as $\mathcal{R}^d_{p,n}$ and, in addition, is $(\mu, c\log(n))$-universal for some constant $\mu > 0$ and a suitable constant $c > 0$. For this purpose we modify $\mathcal{R}^d_{p,n}$ by choosing a variant of Siegel's functions from $\bar{\mathcal{H}}_{n^k,n}$ as secondary hash functions.

DEFINITION 5.5 ($\bar{\mathcal{R}}^{d,k}_{p,n}$). A particular function $h \in \bar{\mathcal{R}}^{d,k}_{p,n}$ is defined by:

- A primary hash function $f \in \mathcal{H}^d_{p,\sqrt{n}}$.
- A secondary hash function $r \circ s$, $r \in \bar{\mathcal{H}}_{n^k,n}$, $s \in \mathcal{H}^1_{p,n^k}$.
- A set of offsets $\mathbf{a} = \langle a_1, \ldots, a_{\sqrt{n}} \rangle$, where $a_i \in [n]$.

The function h is defined in terms of $(f, r \circ s, \mathbf{a})$ as

$$h(x) = (r(s(x)) + a_{f(x)}) \mod n.$$

In the context of $\bar{\mathcal{R}}^{d,k}_{p,n}$, random f means that f is chosen uniformly at random from $\mathcal{H}^d_{p,\sqrt{n}}$, random $r \circ s$ means that r is chosen uniformly at random from $\bar{\mathcal{H}}_{n^k,n}$ and s is chosen uniformly at random from $\mathcal{H}^1_{p,n^k}$, random $\mathbf{a}$ means that, for $i \in [\sqrt{n}]$, a_i is chosen uniformly at random from $[n]$, and random h is defined by random f, random $r \circ s$ and random $\mathbf{a}$. Note that $h \in \bar{\mathcal{R}}^{d,k}_{p,n}$ can be evaluated in constant time if d and k are constants. Further, a random h can be constructed by a randomized DMM with $\sqrt{n}$ processors in constant time. These properties follow directly from Lemma 5.1(a) and (b).

LEMMA 5.3. *Let $S \subseteq U$, $n \le |S| \le n^{11/10}$. In the context of $\bar{\mathcal{R}}^{d,k}_{p,n}$, for each $\ell \ge 0$ there are $d \ge 0$, $k \ge 0$ such that a random pair $(f, r \circ s)$, is d-good for S with probability $1 - n^{-\ell}$.*

PROOF. Let $\ell' > 0$ be given. If d is sufficiently large, then a random f is $2|S|/\sqrt{n}$-perfect on S with probability $1 - n^{-\ell'}$ by Lemma 5.1(e). Note that $r \circ s$ is d-perfect on S if s is 1-perfect on S and r is d-perfect on $s(S)$. Each of these events is true with probability at least $1 - n^{-\ell'}$ by Lemma 5.1(f) and (g). Thus, random $(f, r \circ s)$ is d-good with probability $(1 - n^{-\ell'})^3 \ge 1 - n^{-\ell}$ if ℓ' is sufficiently large relative to ℓ. □

The advantage of $\bar{\mathcal{R}}^{d,k}_{p,n}$ compared with $\mathcal{R}^d_{p,n}$ is its high degree of universality. Whereas $\mathcal{R}^d_{p,n}$ can only be proven to be a $(2, d)$-universal class a much stronger property holds for $\bar{\mathcal{R}}^{d,k}_{p,n}$.

LEMMA 5.4. *Let $\bar{\mathcal{R}}^{d,k}_{p,n}(s)$ be the restriction of $\bar{\mathcal{R}}^{d,k}_{p,n}$ induced by fixing $s \in \mathcal{H}^1_{p,n^k}$. Let $S \subseteq U$, $|S| \le n^{11/10}$. If s is 1-perfect on S, then $\bar{\mathcal{R}}^{d,k}_{p,n}(s)$ is $(1, \sqrt{n})$-universal.*

PROOF. The proof is obvious from Lemma 5.1(d) and the definition of $\bar{\mathcal{R}}^{d,k}_{p,n}$. □

In the next section we analyze properties of our universal hash classes, with respect to a given set S of keys. For this purpose we consider fixed pairs (f, g) that are d-good for S. Thus the random choice of h is reduced to randomly choose the offsets $\mathbf{a}$. As $\mathbf{a}$ consists of $\sqrt{n}$ *independent* random variables, we can apply the following very general tail estimate. It is based on Azuma's inequality. It can be found, e.g., in [1]. The version described below can be found in [2] and [18].

THEOREM 5.1. *Let $X_1, \dots, X_m$ be independent random variables with finite ranges, and let $F(X_1, \dots, X_m)$ be any function in $X_1, \dots, X_m$ with* $\mathrm{Exp}[F] \geq 0$. *Assume that $F(X_1, \dots, X_m)$ only changes by at most an additive offset α in response to a change of one input variable X_i. Then*

$$\mathrm{Prob}[F \geq \mathrm{Exp}[F] + t] \leq e^{-t^2/2\alpha^2 m}.$$

6. The Basic Processes.

The behavior of our PRAM simulations can be modeled by a few simple stochastic processes which we now introduce and analyze. For both Process_1 and Process_2, all results hold with respect to both $\mathcal{R}^d_{p,n}$ and $\bar{\mathcal{R}}^{d,k}_{p,n}$. Since the proofs are so similar for both classes (the only difference is that Lemma 5.2 is used in the proof for $\mathcal{R}^d_{p,n}$ whereas Lemma 5.3 is used in the proof for $\bar{\mathcal{R}}^{d,k}_{p,n}$), we give the proofs only for $\mathcal{R}^d_{p,n}$.

The first process is basic for simulating the read step of a PRAM in our first simulation. Fix $\ell > 0$ arbitrarily and let $S \subseteq U$ be given, where $|S| = n$. For the results with respect to $\mathcal{R}^d_{p,n}$, let d be chosen large enough with respect to ℓ so that a random (f, g) is d-good for S with probability at least $1 - n^{-2\ell}$, and for the results with respect to $\bar{\mathcal{R}}^{d,k}_{p,n}$, let d and k be chosen large enough so that a random $(f, r \circ s)$ is d-good for S with probability at least $1 - n^{-2\ell}$. Let $T = \log\log(n) - 1$ and let $h_1, \dots, h_T$ be functions from U to $[n]$.

Consider the following shrinking process for S.

Process_1

```
For $t = 1$ to $T$ do
    For each $i \in [n]$,
        remove $\min\{4d, |h_t^{-1}(i) \cap S|\}$ elements $x \in h_t^{-1}(i)$ from $S$.
```

THEOREM 6.1. *Let $h_1, \dots, h_T$ be randomly and independently chosen from $\mathcal{R}^d_{p,n}$ (or from $\bar{\mathcal{R}}^{d,k}_{p,n}$). With probability at least $1 - n^{-\ell}$, $|S| \leq n^{9/10}$ at the end of* Process_1.

The proof of this theorem involves several subclaims, which we first develop before giving the proof. Fix $S \subseteq U$, $|S| = n$, and $S' \subseteq S$. Suppose that (f, g) is d-good for S with respect to $\mathcal{R}^d_{p,n}$. For all $i \in [\sqrt{n}]$, define f-bucket

$$B_i = \{S' \cap f^{-1}(i)\},$$

and for all $i \in [\sqrt{n}]$ and $j \in [n]$ define (f, g)-bucket

$$A_{i,j} = \{B_i \cap g^{-1}(j)\}.$$

Let h be defined by $(f, g, \mathbf{a})$. Then the following properties hold:

- For all i, $|B_i| \le 2|S|/\sqrt{n} \le 2\sqrt{n}$.
- For all i, j, $|A_{i,j}| \le d$.
- Independent of $\mathbf{a}$, for all i, j, all keys in $A_{i,j}$ are mapped to the same location by h.
- Independent of $\mathbf{a}$, for all i, for all $j \ne j'$, $h(A_{i,j}) \ne h(A_{i,j'})$, i.e., there is no possible collision between pairs of keys in the same f-bucket but different (f, g)-buckets.
- Let $\mathbf{a}$ be random. For all $i_1, \dots, i_c \in [\sqrt{n}]$, for all $j_1, \dots, j_c \in [n]$,

$$\text{Prob}_{\mathbf{a}}[h(A_{i_1,j_1}) = \cdots = h(A_{i_c,j_c})] \le \frac{1}{n^{c-1}}.$$

LEMMA 6.1. *Fix $S \subseteq U$ with $|S| = n$ and let $S' \subseteq S$ be fixed, $n^{9/10} \le |S'| \le n$. Let (f, g) be d-good for S, let $\mathbf{a}$ be random, and let h be defined by $(f, g, \mathbf{a})$. Let S'' consist of those keys left over from S' after removing* $\min\{4d, |h^{-1}(i) \cap S'|\}$ *elements $x \in h^{-1}(i)$ from S', for each $i \in [n]$. Then*

$$\text{Prob}_{\mathbf{a}}\left[|S''| \ge \frac{|S'|^2}{2n}\right] \le e^{-n^{1/10}/72}.$$

PROOF. Fix any key $x \in S'$, and suppose $x \in A_{i,j}$. Because no (f, g)-bucket has size greater than d, the only way x can fail to be removed is if at least three other (f, g)-buckets map to the same location as where $A_{i,j}$ is mapped. The probability of this event is at most $1/n^3$ times the number of triples of (f, g)-buckets. However, the number of such triples is bounded above by $|S'|^3/6$, and thus

$$\text{Prob}[x \in S''] \le \frac{|S'|^3}{6n^3} \le \frac{|S'|}{6n}.$$

From this it follows that

$$\text{Exp}[|S''|] \le \frac{|S'|^2}{6n}.$$

In order to bound the probability that $|S''|$ is far away from its expectation, we apply the tail estimate Theorem 5.1. Let F be the function which maps the independent random variables $a_1, \dots, a_{\sqrt{n}}$ to $|S''|$. By the definition of d-goodness of (f, g), at most $2\sqrt{n}$ elements $x \in S$ change their hash value $h(x)$ in response to a change of one a_i. The worst effect on $|S''|$ would be that all these elements are noncolliding with respect to S' before the change in a_i but colliding afterward, and thus $|S''|$ can change by at most $2\sqrt{n}$, i.e., the offset α is at most $2\sqrt{n}$. The above theorem now yields

$$\begin{aligned}\text{Prob}\left[|S''| \ge \frac{|S'|^2}{2n}\right] &\le \text{Prob}\left[F \ge \text{Exp}[F] + \frac{|S'|^2}{3n}\right]\\ &\le \exp\left(-\frac{(|S'|^2/(3n))^2}{2(2\sqrt{n})^2 \cdot \sqrt{n}}\right)\end{aligned}$$

$$= \exp\left(-\frac{|S'|^4}{72n^{7/2}}\right)$$
$$\leq \exp\left(-\frac{n^{1/10}}{72}\right) \qquad \text{as} \quad |S'| \geq n^{9/10}.$$

This completes the proof of Lemma 6.1. □

PROOF OF THEOREM 6.1. Let S_t be the set S in Process_1 after t runs of the for loop; S_0 is the initial set S. By Lemma 5.2, the probability that there is a $t \in [T]$ such that (f_t, g_t) is not d-good for S is at most $Tn^{-2\ell} \leq n^{-\ell}/2$ for sufficiently large n. For the remainder of the proof, for all $t \in [T]$, fix (f_t, g_t) to be d-good for S and all probabilities are with respect to random $\mathbf{a}_t$.

Lemma 6.1 implies that, with probability at least $(1 - e^{-n^{1/10}/72})^T$,

$$S_t \leq \max\left\{\frac{|S_{t-1}|^2}{2n}, n^{9/10}\right\} \tag{1}$$

for all $t \in [T]$. The theorem follows by solving the recursion and by observing that the probability there is some $t \in [T]$ for which (1) does not hold is at most $n^{-\ell}/2$ for sufficiently large n. □

We now describe the second process. It is basic for simulating a write step of a PRAM, and also for simulating a read step when we want to come close to time-processor optimality. With respect to the simulation of a write step, the set A_t below denotes the set of unsatisfied write requests stored in the intermediate data structure SM after t steps, compare Section 3.

Let $h\colon U \to [n]$ and $0 < T \leq n^{1/10}$ be fixed. Let $S_t \subseteq U$ be a set of n elements for $1 \leq t \leq T$, $S = \bigcup_{t=1}^{T} S_t$. Fix $\ell > 0$ arbitrarily. For the results with respect to $\mathcal{R}^d_{p,n}$, let d be chosen large enough with respect to ℓ so that a random (f, g) is d-good for S with probability at least $1 - n^{-\ell}/2$ and for the results with respect to $\bar{\mathcal{R}}^{d,k}_{p,n}$, let d and k be chosen large enough so that a random $(f, r \circ s)$ is d-good for S with probability at least $1 - n^{-\ell}/2$.

Process_2

```
$A_0 := \emptyset$.
For $t = 1$ to $T$ do
    $A_t := A_{t-1} \cup S_t$
    For $i \in [n]$ let $C_i$ be the set of keys $x \in A_t$ with $h(x) = i$.
        Remove from $A_t$ $\min\{4, |C_i|\}$ keys $x \in C_i$.
```

Note that there is a more involved analysis that shows essentially the same result as stated in the following theorem when only two elements (as opposed to four) are removed from each C_i.

THEOREM 6.2. *For h randomly drawn from $\mathcal{R}^d_{p,n}$ or $\bar{\mathcal{R}}^{d,k}_{p,n}$, with probability at least $1 - n^{-\ell}$,* Process_2 *fulfills $|A_t| \leq 2d^2 n$ for all $t \in [T]$.*

PROOF. By Lemma 5.2 a random (f, g) fails to be d-good for S with probability at most $n^{-\ell}/2$. For the remainder of the proof we fix (f, g) to be d-good for S, and all probabilities and expectations are with respect to random $\mathbf{a}$. Let h be defined by $(f, g, \mathbf{a})$.

CLAIM 6.1. *For all* $t \in [T]$, $\operatorname{Exp}[|A_t|] \leq d^2 n$.

PROOF. In [8] (see Definition 5.5 and Theorem 6.1) the following is shown.

LEMMA 6.2. *Let* $S \subseteq U$, $|S| \leq tn$ *for some* $t \geq 1$. *Let* h *be as above*; h *splits* S *into buckets* $B_1, \ldots, B_n$. *Then, for any* $u \geq 4$ *and for each* $i \in [n]$,

$$\operatorname{Prob}[|B_i| \geq ut] \leq 2^{-ut/d}.$$

If, at some step t, $|C_i| \geq u$, then there is a $t' \leq t$ such that $S'_{t'} = \bigcup_{l=t-t'+1}^{t} S_l$ contains a set $B'_{t'}$ of at least $u + 4t'$ keys which are mapped to i by h. From Lemma 6.2,

$$\begin{aligned} \operatorname{Prob}[|C_i| \geq u] &\leq \sum_{t' \leq t} \operatorname{Prob}[B'_{t'} \geq u + 4t'] \\ &\leq \sum_{t' \leq t} 2^{-(u+4t')/d} = 2^{-u/d} \sum_{t' \leq t} 2^{-4t'/d} \leq \frac{d2^{-u/d}}{2}. \end{aligned} \tag{2}$$

Thus

$$\operatorname{Exp}[|C_i|] \leq \sum_{u \geq 0} \operatorname{Prob}[|C_i| \geq u] \leq d^2,$$

and

$$\operatorname{Exp}[|A_t|] = \sum_{i \in [n]} \operatorname{Exp}[|C_i|] \leq d^2 n.$$

This completes the proof of Claim 6.1. □

We now apply the tail estimate Theorem 5.1 to the function which maps S according to a random $\mathbf{a}$. By the definition of goodness, at most $2|S|/\sqrt{n} \leq 2n^{3/5}$ keys from S are affected by the change of one a_i. The worst effect on $|A_t|$ would be that none of these keys are in A_t before the change of a_i, but all of them are afterwards. Thus $\alpha \leq 2n^{3/5}$. Theorem 5.1 now yields

$$\operatorname{Prob}[|A_t| \geq 2d^2 n] \leq \operatorname{Prob}[|A_t| \geq \operatorname{Exp}[|A_t|] + d^2 n] \leq e^{-d^4 n^{3/10}/8}.$$

Thus, the probability that there is a $t \in [T]$ such that $|A_t| \geq 2d^2 n$ is at most $Te^{-d^4 n^{3/10}/8}$ which is at most $n^{-\ell}/2$ for sufficiently large n. This completes the proof of Theorem 6.2. □

Inequality (2) shown in the proof of Claim 6.1 immediately implies the following remark.

REMARK 6.1. At the end of Process_2, $|C_i| = O(\log(n))$ for all $i \in [n]$, with probability at least $1 - n^{-\ell}$.

For our time-processor optimal simulation we need an extension of Process_2 which we call Extended_Process_2. In the Extended_Process_2, after each group of $\log\log(n)$ rounds of Process_2 a *size-reduction* is executed. In the size-reduction, $\min\{2d \log\log(n), |C_i|\}$ keys from C_i are removed from A, for $i = 1, \dots, n$.

THEOREM 6.3. *In addition to the property from Theorem* 6.2, *the following holds for the* Extended_Process_2. *After each size-reduction,* $|A| = O(n/\log(n))$ *with probability at least* $1 - n^{-\ell}$.

PROOF. By Lemma 5.2 a random (f, g) fails to be d-good for S with probability at most $n^{-\ell}/2$. For the remainder of the proof we fix (f, g) to be d-good for S, and all probabilities and expectations are with respect to random $\mathbf{a}$. Let A_t be as in the last proof. Assume that a size-reduction is executed in the Extended_Process_2 after t steps. Let $C_i \subseteq A_t$ be the set of key from A_t mapped to i by h. As shown in the proof of Lemma 6.2, $\text{Prob}[|C_i| \geq u] \leq d2^{-u/d}/2$. Thus, $\text{Exp}[|\{i, |C_i| \geq 2d \log\log(n)\}|] \leq dn/\log^2(n)$. A further application of the tail estimate Theorem 5.1 shows that

$$\text{Prob}\left[|\{i, |C_i| \geq 2d \log\log(n))| \geq \frac{2dn}{\log(n)^2}\right] \leq \frac{n^{-\ell}}{4}$$

for sufficiently large n. Also,

$$\text{Prob}[\max_{1 \leq i \leq n} |C_i| \geq 3\ell d \log(n)] \leq \frac{n^{-\ell}}{4}$$

for sufficiently large n can be easily concluded. For all $i \in [n]$, the reduction phase removes $\min\{|C_i|, 2d \log\log(n)\}$, and thus at most $6\ell d^2 n/\log(n)$ elements are in A_t after the size reduction, with the desired probability. □

We now describe the third process. It is basic for simulating a read step of a PRAM in the last three simulations. For this process we need the stronger universal properties of $\bar{\mathcal{R}}^{d,k}_{p,n}$ for the analysis. Let $S \subset U$ be a set of $n/16$ keys. Let h_1 and h_2 map U into $[n]$.

Process_3

```
Repeat until $S = \emptyset$
  For $j = 1, 2$ do
      For $i \in \{1, \dots, n\}$
          remove one $x \in h_j^{-1}(i)$ from $S$
```

Let $\ell > 0$. Let d and k be sufficiently large constants relative to ℓ so that a random s is 1-perfect for S with probability at least $1 - n^{-\ell}/4$.

THEOREM 6.4. *Let* h_1 *and* h_2 *be randomly and independently chosen from* $\bar{\mathcal{R}}^{d,k}_{p,n}$. *With probability at least* $1 - n^{-\ell}$, *the repeat until loop of* Process_3 *is executed at most* $D \log\log(n)$ *times before* $S = \emptyset$ *for some constant* D.

PROOF. Fix s_1 and s_2 so that they are both 1-perfect on S. All probabilities in the following are with respect to random $h_1 \in \bar{\mathcal{R}}_{p,n}^{d,k}(s_1)$ and $h_2 \in \bar{\mathcal{R}}_{p,n}^{d,k}(s_2)$. Consider the following directed labeled graph $G = \{[n], E\}$ (with multi-edges and self-loops allowed) defined by h_1, h_2, and S. There is an edge from $h_1(x)$ to $h_2(x)$ labeled x for each $x \in S$. This graph has the structural properties stated in Lemma 6.3 and Claim 6.2 below. Similar properties are well known for random graphs [3] and are proved using similar techniques. Note that both h_1 and h_2 are $(1, \sqrt{n})$-universal on S because of Lemma 5.4.

LEMMA 6.3. *Let H be the graph obtained from G by removing all labels and directions from the edges. For each $\ell \geq 1$ there are $\beta, s \geq 1$ such that:*

(a) Prob[*H has a connected component of size at least $\beta \log(n)$*] $\leq n^{-\ell}/4$
(b) Prob[*H has a connected component A with at least $|A| + s - 1$ edges*] $\leq n^{-\ell}/4$.

PROOF. The proof of the lemma relies on the following claim.

CLAIM 6.2. *Let $k \geq 2$, $s \geq 0$, $k + s - 1 \leq \sqrt{n}$. The probability there is a subgraph $G' \subseteq G$ such that G' contains k vertices and at least $k + s - 1$ edges is at most*

$$n^{-s+1} \cdot (k+1)^{s-1} \cdot 2^{-(k+2s-5)}.$$

PROOF. Let $\mathcal{G}_{k,s}$ be the set of all directed labeled (with elements of S) graphs on node set $[n]$ with k vertices and $k + s - 1$ edges. Then,

$$\begin{aligned}|\mathcal{G}_{k,s}| &\leq \binom{n}{k} \cdot \binom{k^2+k+s-1}{k+s-1} \cdot \left(\frac{n}{16}\right)^{k+s-1} \\ &\leq n^{2k+s-1} \cdot e^{2k+s} \cdot (k+1)^{s-1} \cdot 2^{-4(k+s-1)}.\end{aligned}$$

Because $k + s - 1 \leq \sqrt{n}$ and h_1, h_2 are independently chosen from a $(1, \sqrt{n})$-universal class of hash functions the following is true. For a fixed $G' \in \mathcal{G}_{k,s}$, for randomly chosen h_1 and h_2, the probability that the directions and labels with respect to h_1 and h_2 coincide with G', i.e., the probability that G' is a subgraph of G, is at most $n^{-2(k+s-1)}$. Therefore, the probability there is some $G' \in \mathcal{G}_{k,s}$ such that G' is a subgraph of G is at most

$$n^{-s+1} \cdot (k+1)^{s-1} \cdot 2^{-(k+2s-5)}.$$

This complete the proof of Claim 6.2. □

Now we complete the proof of Lemma 6.3. Part (a) follows from Claim 6.2 with $k = \beta \log(n)$ for β a sufficiently large constant and $s = 0$. Part (b) follows from Claim 6.2 and by applying part (a) and then applying Claim 6.2 for s fixed to a sufficiently large contant and using all values of $k \in [\beta \log(n)]$. □

To finish the proof of Theorem 6.4, we translate the effect of the repeat until loop into a game on the graph H: Each run of the loop corresponds to removing, for each nonisolated node of H, an incident edge. The end of the loop corresponds to the situation where H has lost all its edges.

Consider a connected component of H with vertex set A, $|A| = k$, and edge set $E(A)$, $|E(A)| = k + s - 1$. Let $E'(A) \subseteq E(A)$ form a spanning tree of this component, $|E'(A)| = k - 1$. We consider our game on H restricted to A.

There are at most s moves in which edges from $E(A) \backslash E'(A)$ are removed. The other moves only remove edges of the spanning tree. In each of these moves at least half of the remaining edges of the spanning tree are removed, i.e., $\log(k)$ moves suffice to remove all these edges. Thus all edges of the component are removed after $\log(k) + s$ moves. As, by Lemma 6.3, for each component, $k = O(\log(n))$ and $s = O(1)$ holds with high probability the theorem follows. □

7. Two Fast Simulations. We present two simulations of an n processor PRAM on an n processor DMM. The first simulation has delay and work (i.e., overall number of operations executed by the DMM to simulate one PRAM step) $\Theta(\log\log(n)\log^*(n))$ with high probability. Thus it cannot be directly converted into a time-processor optimal simulation. It has the advantage that it uses hash functions from $\mathcal{R}^d_{p,n}$ rather than the more complicated class $\bar{\mathcal{R}}^{d,k}_{p,n}$.

The second simulation is faster; its delay is only $O(\log\log(n))$. More importantly, it only uses optimal work $O(n)$ for simulating one step of the PRAM with high probability. It is the basis for the time-processor optimal simulation. Its disadvantage is that we need the more complicated class of hash functions $\bar{\mathcal{R}}^{d,k}_{p,n}$ in order to make our analysis work.

We show how to simulate a *phase* of up to $n^{1/10}$ steps of the PRAM. After a phase, we perform a cleanup step which consists of dumping all the data currently stored in the temporary shared memories into their final locations. After the cleanup step, all temporary shared memories are empty for the start of the next phase of the simulation. The purpose of the cleanup step is to ensure that all temporary shared memories are of size $O(n)$ at all points in time with high probability. Remark 6.1 shows that the cleanup step takes time $O(\log(n))$ per temporary shared memory, with high probability.

Let $S \subseteq U$, $|S| \leq n^{11/10}$, denote the set of keys used as shared memory addresses in the phase of the PRAM to be simulated. Both simulations use an algorithm WRITE, which in turn uses a hash function h and a perfect hash table SM (see Section 4). The hash function h is randomly chosen from $\mathcal{R}^d_{p,n}$ or $\bar{\mathcal{R}}^{d,k}_{p,n}$ for suitable $d, k > 0$. During the simulation, the name of each shared memory cell x, together with its current content $c(x)$, is stored in SM or in a module, such that the following holds:

INVARIANT. At each time t, for each shared memory cell $x \in U$ for which $c(x)$ has been defined at time t, $(x, c(x))$ is either stored in SM, or, if not, in $M_{h(x)}$.

Hereafter, for brevity we refer to names x of shared memory cells as keys, and we write x instead of $(x, c(x))$. Recall that we use SM to refer to both the name of the perfect hash table and its contents.

Let $X = \{x_1, \ldots, x_n\}$ denote the keys to be written during a PRAM write step. In case of an EREW PRAM $x_1, \ldots, x_n$ are distinct, in case of a CRCW PRAM this is not necessary, i.e., X may be a multiset. For $i = 1, \ldots, n$ let x_i be associated with processor P_i. The first step is to add X to SM using the algorithm HASH described in Section 4. The second step attempts to simultaneously move the keys in SM into the memory modules.

Recall that a temporary perfect hash table SM is distributed among the memory modules, with at most c entries of SM stored in each module for some constant c. Let D be an integer. The algorithm WRITE_M(SM, h, D) moves

$$\min\{|\mathrm{SM} \cap h^{-1}(i)|, D\}$$

keys $x \in \mathrm{SM} \cap h^{-1}(i)$ from SM to memory module $M_{h(x)}$, for all $i = 1, \ldots, n$. This is implemented as follows. Simultaneously, for all $i \in [n]$, processor P_i reads in sequence the at most c entries of SM stored in memory module M_i. Then, simultaneously, for all $i \in [n]$, P_i tries to write in sequence, for $j = 1, \ldots, c$, the jth key x to memory module $M_{h(x)}$ up to D times. Each memory module can accept up to one write request per time step. This takes time $O(c \cdot D)$, i.e., constant time.

```
**WRITE**($h$, SM, $X$)
    HASH(SM, $X$)
    If "failure" returned then call EMERGENCY_WRITE($h$, SM, $X$)
    WRITE_M(SM, $h$, 4)
**EMERGENCY_WRITE**($h$, SM, $X$)
    WRITE_M(SM $\cup$ $X$, $h$, $\infty$)
```

LEMMA 7.1. WRITE *satisfies the following*:

(a) WRITE *fulfills the invariant.*
(b) WRITE *runs within time* $O(\log^*(n))$ *with high probability.*

PROOF. (a) is immediate from the description.

(b) is immediate from the description and the analysis of HASH, if EMERGENCY_WRITE is not invoked. EMERGENCY_WRITE takes time $O(n)$ in the worst case. Therefore, (b) is implied by the following claim.

CLAIM 7.1. *Let* $A_t \subset U$ *be the contents of* SM *after* t *simulated steps. There is a constant* $c > 0$ *such that for each* t, $1 \le t \le n^{1/10}$, $|A_t| \le cn$ *with high probability.*

PROOF. Observe that the additions and deletions from SM during the algorithm are captured by Process_2. The claim now follows from Theorem 6.2. □

The following two PRAM simulations use WRITE as a subroutine. The first simulation uses $\log\log(n)$ hash functions $h_1, h_2, \ldots, h_{\log\log(n)}$ chosen randomly and independently from $\mathcal{R}^d_{p,n}$, to store $\log\log(n)$ copies of each PRAM cell. The jth copy of cell x is to be found either in SM_j or in module $M_{h_j(x)}$. A read step consists of $\log\log(n)$ iterations. At the jth iteration, each processor that has not yet succeeded in reading looks for its key x first in SM_j and then, if it has still not succeeded, in $M_{h_j(x)}$. The second simulation is similar, but uses only two hash functions, h_1 and h_2, and hence only two copies of each cell. The read step again consists of $\log\log(n)$ iterations. At each iteration, each processor that has not yet succeeded tries to access both copies of the cell it is seeking. Our analysis of this second algorithm requires that the hash functions be drawn from the more complicated family $\bar{\mathcal{R}}^{d,k}_{p,n}$.

7.1. *A Simulation with Nonoptimal Work.* This simulation stores each shared memory cell in $\ell = \log\log(n)$ modules, specified by ℓ hash functions from $\mathcal{R}^d_{p,n}$, for suitable d. We use ℓ shared memories $\mathrm{SM}_1, \ldots, \mathrm{SM}_l$ of size $c'n$, each.

Let $S \subseteq U$, $|S| \leq n^{11/10}$ be the set of keys used in a phase of $n^{1/10}$ PRAM steps. Each PRAM step consists of two substeps: a write step followed by a read step. We let $X = \{x_1, \ldots, x_n\}$ denote the multiset of n keys that are to be written or read during a particular PRAM step. We use the algorithm LOOKUP described in Section 4. Let READ(x, M, ans) indicate a read request to module M for key x. If M has many simultaneous read requests to different keys, it can only successfully complete one of them. It returns $ans = ok$ for the successful read request and $ans = fail$ for all the unsuccessful read requests. If Read(x, M, ans) is called by several processors (as is allowed in the CRCW PRAM), then either all of them or none of them are successful.

```
SIMULATION_1

PREPROCESSING_1
  Choose $h_1, \ldots, h_\ell$ randomly and independently from $\mathcal{R}^d_{p,n}$.

WRITE_1($X$)
  For $j = 1, \ldots, \ell$, WRITE($h_j$, SM$_j$, $X$)

READ_1($X$)
  (i) For all $i \in [n]$, status($i$) := "failure."
      For $j = 1, \ldots, \ell$, LOOKUP(SM$_j$, $X$).
        Simultaneously, for all $i \in [n]$,
          If contents of $x_i$ found in SM$_j$
          then status($i$) := "success"
 (ii) For $j = 1, \ldots, \ell - 1$
        Repeat $D$ times
          Simultaneously, for all $i \in [n]$
            If status($i$) = "failure" then READ($x_i$, $M_{h_j(x_i)}$, $ans$)
            If $ans = ok$ then status($i$) := "success"
(iii) Repeat until, for all $i \in [n]$, status($i$) = "success"
        Simultaneously, for all $i = 1, \ldots, n$
          READ($x_i$, $M_{h_\ell(x_i)}$, $ans$)
          If $ans = ok$ then status($i$) := "success"
```

THEOREM 7.1. SIMULATION_1 *simulates an n processor CRCW-PRAM on an n processor* DMM *using delay* $O(\log\log(n)\log^*(n))$ *with high probability.*

PROOF. It is clear that the above algorithm correctly simulates a PRAM. PREPROCESSING_1 runs within time $O(\log\log(n))$ in the worst case. From Lemma 7.1, it follows that WRITE_1 runs within time $O(\log\log(n)\log^*(n))$ with high probability.

Each run of each of the above three loops within READ_1 take constant time. Thus (i) and (ii) take time $O(l) = O(\log\log(n))$. Loop (ii), which tries to find keys in the modules according to the hash functions $h_1, \ldots, h_{\ell-1}$, follows the rules of Process_1. Thus, at the end of loop (ii), at most a set X', $|X'| \le n^{9/10}$, has not obtained an answer, i.e., their corresponding read requests are not yet satisfied, with high probability. This is shown in Theorem 6.1.

In [8], it is shown that a random $h_\ell \in \mathcal{R}^d_{p,n}$ is d-perfect (for a sufficiently large constant D) on a set X' of size at most $n^{9/10}$, with high probability. Thus loop (iii) is finished after D rounds, with high probability. □

It is easily verified that SIMULATION_1 can not be converted into a time-processor optimal simulation, because it needs work $\Omega(n\log\log(n))$ for simulating one step of the PRAM. The reason for this is that we use a nonconstant number of hash functions.

7.2. *A Simulation with Optimal Work.* We now present a simulation that uses only two hash functions, but we have to choose them from the more complicated class $\bar{\mathcal{R}}^{d,k}_{p,n}$ for suitably chosen constants d and k. We use two shared memories, SM_1 and SM_2. Again let $X = \{x_1, \ldots, x_n\}$ be the multiset of keys requested by the PRAM step. The new algorithm for reading now proceeds like Read_1, except that it does not need loop (iii). Instead, it alternates between the two hash functions in loop (ii). For technical reasons, we satisfy the read requests to the modules in 16 batches of $n/16$ keys, each.

```
SIMULATION_2
PREPROCESSING_2
  Choose $h_1, h_2$ randomly and independently from $\bar{\mathcal{R}}^{d,k}_{p,n}$.
WRITE_2($X$)
  For $j = 1, 2$, WRITE($h_j$, SM$_j$, $X$)
READ_2($X$)
 (i) For all $i \in [n]$, status($i$) := "failure."
     For $j = 1, 2$ do LOOKUP(SM$_j$, $X$)
        Simultaneously, for all $i \in [n]$,
           If contents of $x_i$ found in SM$_j$
           then status($i$) := "success"
 (ii) For $t = 0, \ldots, 15$ do
        $u := 1 + tn/16$, $v = (t+1)n/16$
        Repeat until, for all $i \in \{u, \ldots, v\}$, status($i$) = "success"
           for $j = 1, 2$
              Simultaneously, for all $i \in \{u, \ldots, v\}$
                 If status($i$) = "failure" then READ($x_i$, $M_{h_j(x_i)}$, $ans$)
                 If $ans = ok$ then status($i$) := "success"
```

THEOREM 7.2. SIMULATION_2 *simulates an n processor CRCW PRAM on an n processor* DMM *using delay* $O(\log\log(n))$ *with high probability.*

PROOF. Clearly the above algorithm correctly simulates a PRAM. PREPROCESSING_2 runs in constant time. From Lemma 7.1, WRITE_2 runs in time $O(\log^*(n))$ with high probability. The LOOKUPs in SM_1 and SM_2 within READ_2 each take constant time. Because the loop within READ_2 follows the rules of Process_3, it follows from Theorem 6.4 that, when D is chosen to be a sufficiently large constant, for each value of $t = 0, \ldots, 15$, the number of iterations of this loop is $D \log\log(n)$ with high probability. □

It can easily be checked that this simulation uses optimal work $O(n)$ to simulate a step of the PRAM, with high probability.

8. Fast and Almost Optimal Simulation.

In this section we present a simulation of an $n\log\log(n)$ processor CRCW PRAM on an n- processor DMM. The simulation achieves almost optimal delay $O((\log\log(n)\log^*(n))$, with high probability. Let $\ell = \log\log(n)$. Assume that the PRAM-processors are grouped into n blocks of l processors each. The DMM has processors $Q_1, \ldots, Q_n$, where, for $i = 1, \ldots, n$, Q_i simulates the ith block. For $k = 1, \ldots, \ell$, we let X_k denote the multiset of n keys, the kth key from each block, to be written or read during a particular PRAM step. We use three hash functions h_0, h_1, h_2, three intermediate hash tables $\mathrm{SM}_0, \mathrm{SM}_1, \mathrm{SM}_2$, and one additional temporary hash table $\overline{\mathrm{SM}}$ for each execution of the read algorithm. All of the hash tables are maintained using the algorithm HASH described in Section 4.

The interesting aspect of this algorithm is how the read requests are processed by READ_3. Step (i) processes the read requests associated with keys currently stored in SM_0, SM_1, or SM_2. All the read requests successfully processed in (i) require no further processing in (ii). In step (ii) most of the remaining read requests are successfully satisfied within the loop using hash function h_0. $\overline{\mathrm{SM}}$ is used to store the unsatisfied read requests during the execution of this loop. In step (iii) the remaining $O(n)$ unsatisfied read requests in $\overline{\mathrm{SM}}$ are then satisfied using h_1 and h_2.

The algorithm D_READ_M($\overline{\mathrm{SM}}$, h_0) removes, for all $i \in [n]$,

$$\min\{|\overline{\mathrm{SM}} \cap h_0^{-1}(i)|, 4\}$$

keys from $\overline{\mathrm{SM}}$, and sends these read requests to the appropriate module, i.e., the request for key x is sent to module $M_{h_0(x)}$. We say that these read requests have been satisfied. $M_{h_0(x)}$ does not immediately send back the value of the key $c(x)$ associated with x (The "D" in "D_READ" stands for "delayed"); instead it maintains a list of read requests it has promised to process, and delays sending back the values until RESPOND_REQUESTS is executed in step (iv). Clearly, this algorithm runs in constant time if $|\overline{\mathrm{SM}}| = O(n)$.

The loop within step (ii) is similar to WRITE: At the beginning of iteration k, the read requests that have not been satisfied among $X_1 \cup \cdots \cup X_{k-1}$ are currently residing in $\overline{\mathrm{SM}}$. First, X_k is added to $\overline{\mathrm{SM}}$ using HASH, and then D_READ_M($\overline{\mathrm{SM}}$, h_0) is executed to ensure that $|\overline{\mathrm{SM}}| = O(n)$ at all times within the loop. The fact that $\overline{\mathrm{SM}}$ stays small follows from how Process_2 works.

At the termination of step (ii), $\overline{\text{SM}}$ still stores up to $O(n)$ read requests that have not been satisfied. In step (iii) D_READ_2 is used to process these remaining read requests using hash functions h_1 and h_2 exactly how READ_2 works, except that once again the only immediate action of the modules is to acknowledge which of the read requests they will process, and they delay sending back the values until RESPOND_REQUESTS is executed in step (iv). EMERGENCY_READ is analogous to EMERGENCY_WRITE described above.

```
SIMULATION_3
PREPROCESSING_3
  Choose $h_0, h_1, h_2$ randomly and independently from $\bar{\mathcal{R}}^{d,k}_{p,n}$
WRITE_3($X_1, \ldots, X_\ell$)
  For $j = 0, 1, 2$
    For $k = 1, \ldots, \ell$, WRITE($h_j$, SM$_j$, $X_k$)
READ_3($X_1, \ldots, X_\ell$)
 (i) For $j = 0, 1, 2$
       For $k = 1, \ldots, \ell$, LOOKUP(SM$_j$, $X_k$)
 (ii)  $\overline{\text{SM}} := \emptyset$
       For $k = 1, \ldots, \ell$
         HASH($\overline{\text{SM}}$, $X_k$)
         If "failure" then call EMERGENCY_READ($h_0$, $\overline{\text{SM}}$, $X_k$)
         D_READ_M($\overline{\text{SM}}$, $h_0$)
 (iii) D_READ_2($\overline{\text{SM}}$) (Using $h_1$ and $h_2$)
 (iv) RESPOND_REQUESTS.
```

In the algorithm RESPOND_REQUESTS, the modules finally send back the values for the read requests to the issuing processors they have promised to answer during the execution of D_READ_M and D_READ_2. This is done as follows: When READ_3 is executed, a key may move from one memory module to another, i.e., when HASH is executed and $\overline{\text{SM}}$ is reformed. During this time, each module that receives a key saves a pointer indicating where the key came from. When RESPOND_REQUESTS is finally executed, this trail of pointers indicating the path of key x is followed in a pipeline fashion back to the version of $\overline{\text{SM}}$ to which x was added by an execution of HASH. At this point, all processors that have requested x at this time execute LOOKUP(x, $\overline{\text{SM}}$) for this version of $\overline{\text{SM}}$. Since each key moves $O(\log\log(n)\log^*(n))$ times, the total time for pipelining back all of the values to the originating processors takes time $O(\log\log(n)\log^*(n))$.

THEOREM 8.1. SIMULATION_3 *simulates an* $n\log\log(n)$ *processor CRCW PRAM on an n processor* DMM *using delay* $O(\log\log(n)\log^*(n))$ *with high probability.*

PROOF. The above algorithm clearly simulates a CRCW-PRAM correctly. PREPROCESSING_3 runs in constant time. WRITE_3 runs in time $O(\log\log(n)\log^*(n))$ with high probability.

In READ_3, step (i) runs in time $O(\log\log(n))$ in the worst case. There are $O(\log\log(n))$

iterations of the loop in step (ii), and each iteration runs in time $O(\log^*(n))$ with high probability, if EMERGENCY_READ is not invoked. (This time is governed by the time for reforming $\overline{\mathrm{SM}}$ using HASH, and this time is $O(\log^*(n))$ with high probability.) As the manipulation of $\overline{\mathrm{SM}}$ during the loop follows the rules of Process_2, Theorem 6.2 guarantees that EMERGENCY_READ is not invoked with high probability. In step (iii), the execution of D_READ_2 runs in time $O(\log\log(n))$ with high probability as shown in the analysis of SIMULATION_2. The time for step (iv) is dominated by the time for steps (ii) and (iii). □

9. A Fast, Optimal Simulation.

To describe the fast and optimal simulation, we first introduce one more hashing technique.

9.1. *Simultaneous Hashing.*

For the time-processor optimal simulation we need a highly efficient implementation of a data structure called an *approximate compaction table* which is less powerful than a parallel hash table. An approximate compaction table stores a set of up to cn key-value pairs in a table with $c'n$ cells, each of which is capable of storing a key-value pair. Unlike a parallel hash table, an approximate compaction table is not required to support the LOOKUP operation.

The basic operation that we require is called SIMULTANEOUS-HASH. To describe this operation we require some preliminary definitions. Let the set of n processor-module pairs (P_i, M_i) be partitioned into $\log^*(n)$ sets of cardinality $n/\log^*(n)$. Let the rth of these sets be denoted DMM^r. For each r, let Y^r be a set of key-value pairs stored in the memory modules of DMM^r, such that at most $\log^*(n)$ keys of Y^r reside in any memory module. The elements of Y^r that a module contains are stored in an array within the module. For each r, let SM^r be a parallel hash table of size $c'n$ distributed among the n memory modules, occupying c' cells in each module. The constant c' will be specified below.

SIMULTANEOUS-HASH(SM, Y)

Input: A collection $\mathrm{SM} = \{\mathrm{SM}^r\}$ of approximate compaction tables and a collection $Y = \{Y^r\}$ of sets of key-value pairs, where r ranges over $[\log^*(n)]$.
Result: For each r, the following holds: if $|\mathrm{SM}^r \cup Y^r|$ exceeds cn, then the value "full" is returned; otherwise, SM^r is augmented by the insertion of the set Y^r.

We now give an algorithm to perform SIMULTANEOUS-HASH(SM, Y). The set of processors of the DMM is partitioned into $\log^*(n)$ subsets, each of size $n/\log^*(n)$; the rth subset is denoted DMM^r. For $r = 1, 2, \ldots, \log^*(n)$, a $c'n/\log^*(n) \times \log^*(n)$ array $\overline{\mathrm{SM}}^r$ is set up in the memory modules of DMM^r, where the constant c' is large enough so that $2cn/\log^*(n)$ keys can be stored in a parallel hash table of size $c'n/\log^*(n)$ in time $O(\log^*(n))$ with failure probability at most $n^{-\ell}$. Each of these $n/\log^*(n)$ memory modules holds c' rows of the array.

For all $r = 1, \ldots, \log^*(n)$, the contents of SM^r is copied into $\overline{\mathrm{SM}}^r$. The elements of Y^r are then inserted into $\overline{\mathrm{SM}}^r$; this process will fail if $|\mathrm{SM}^r \cup Y^r| > cn$. Finally, if the process succeeds, the new contents of the array $\overline{\mathrm{SM}}^r$ is copied into SM^r.

In order to perform the first step, an arbitrary but fixed one-to-one correspondence is established between the cells of SM^r and the cells of $\overline{\mathrm{SM}}^r$. The copying operation can then be scheduled to execute in time $O(\log^*(n))$, since each module sends only $O(\log^*(n))$ key-value pairs (at most c' keys from each parallel hash table SM^r) and receives only $O(\log^*(n))$ key-value pairs. The third step is performed similarly.

We now describe the second step, in which, simultaneously for all r, the set Y^r of key-value pairs is inserted into $\overline{\mathrm{SM}}^r$, using the processors and modules in DMM^r. For each r this is done as follows.

- In each row of $\overline{\mathrm{SM}}^r$ the processor associated with that row moves its keys to the rightmost positions in the row.
- For each i, let the key (if any) in the first cell of the ith row of $\overline{\mathrm{SM}}^r$ be a_i, and let the key (if any) in the second cell of the ith row of $\overline{\mathrm{SM}}^r$ be b_i. Let

$$S = \left\{a_i, b_i : i \in \left[\frac{c'n}{\log^*(n)}\right]\right\}.$$

  Using the $O(\log^*(n))$-time parallel hash table algorithm, store S in a parallel hash table T of size $c'n/\log^*(n)$. It can be verified that if $|\overline{\mathrm{SM}}^r| \le cn$, then $|S| \le 2cn/\log^*(n)$, and thus by the choice of c' this step fails with probability at most $n^{-\ell}$.
- For all i: if a_i is stored in $T[j]$n then the processor associated with row i copies the first cell of row i into the second cell of row j; For all i: if b_i is stored in $T[j]$ then the processor associated with row i copies the second cell of row i into the second cell of row j.
- The set Y^r is copied to the first column of $\overline{\mathrm{SM}}^r$.

THEOREM 9.1. SIMULTANEOUS-HASH(SM, Y) *can be performed within time* $O(\log^*(n))$, *with high probability.*

PROOF. The first and third steps run in worst-case time $O(\log^*(n))$. We now show that the second step operates within the required time bound with sufficiently high probability, provided that c' is chosen large enough. The step consists of four substeps. The first, third, and fourth substeps terminate within worst-case time $O(\log^*(n))$. By the choice of c', the second step fails with probability at most $n^{-\ell}$. □

9.2. *The Simulation.* In this subsection we present a simulation of an $n \log\log(n) \log^*(n)$ processor EREW PRAM on an n processor DMM which achieves optimal delay

$$O(\log\log(n) \log^*(n))$$

with high probability.

Assume that the PRAM processors are $P_{i,k}$, $i \in [n]$, $k \in [\log\log(n)\log^*(n)]$. The processors of the DMM are $Q_1, \ldots, Q_n$, where Q_i simulates

$$P_{i,1}, \ldots, P_{i,\log\log(n)\log^*(n)},$$

for $i = 1, \ldots, n$. We again use three hash functions h_0, h_1, h_2. $Q_1, \ldots, Q_n$ are partitioned into $\log^*(n)$ groups $G_1, \ldots, G_{\log^*(n)}$, each of size $n/\log^*(n)$. In the same way, $M_1, \ldots, M_n$ are partitioned into groups $H_1, \ldots, H_{\log^*(n)}$.

Let $\ell = \log\log(n)$. Let $X = (\bigcup_{s\in[\ell], r\in[\log^*(n)]} X_{r,s}) \subseteq U$, $X_r = \bigcup_{s\in[\ell]} X_{r,s}$ is stored in G_r, each processors of G_r has stored $\log^*(n)$ keys from each $X_{r,s}$, $s = 1, \dots, \log\log(n)$. Let $X^s = \bigcup_{r\in[\log^*(n)]} X_{r,s}$, for $s = 1, \dots, \ell$.

The idea of the simulation is as follows: Let SIMULATION_3 run separately for each group G_r. In the sth round, the n keys from $X_{r,s}$ are accessed. A naive implementation would need time $O(\log\log(n)(\log^*(n))^2))$ altogether, with high probability. We see that the algorithm SIMULTANEOUS-HASH can be used to reduce this runtime by a $\log^*(n)$ factor.

For each h_j, $j = 0, 1, 2$, we use a $c'n\log^*(n)$-array SM_j with columns SM_j^r, $r = 1, \dots, \log^*(n)$. As SIMULTANEOUS-HASH is not able to eliminate duplicates of keys, we have to assume that all keys used in a PRAM-step are different, i.e., that we simulate an EREW PRAM. At the end of the write phase the (at most cn) keys from SM_j are stored in a parallel hash table GM_j of size $c'n$. For reading, we use the $c'n \times \log^*(n)$ array SM with columns SM^r, $r = 1, \dots, \log^*(n)$, and a further hash table $\widetilde{GM}$ of size $c'n$.

```
SIMULATION_4

PREPROCESSING_4
        Choose $h_0, h_1, h_2$ uniformly and independently from $\bar{\mathcal{R}}_{p;n}^{d,k}$ for suitable $d$,
        $k > 0$.
WRITE_4($X_1, \dots, X_\ell$)
      For $j = 0, 1, 2$
      (i)For $s = 1, \dots, \ell$
            SIMULTANEOUS-HASH($\mathrm{SM}_j$, $X_s$)
            If "failure" then EMERGENCY_WRITE($h_j$, $\mathrm{SM}_j$, $X_s$)
            For $r = 1, \dots, \log^*(n)$, WRITE_M($\mathrm{SM}^r{}_j$, $h_j$, $D$)
        Comment: At the end of loop (i), each $\mathrm{SM}^r{}_j, r = 1, \dots, \log^*(n)$, has
        size $O(n)$ with high probability. The next loop reduces each $\mathrm{SM}^r{}_j$ to size
        $O(n/\log(n))$, with high probability.
(ii)    For $r = 1, \dots, \log^*(n)$, WRITE_M($\mathrm{SM}^r{}_j$, $h_j$, $\log\log(n)$)
        HASH($\mathrm{SM}_j$, $GM_j$, $full$)
        If $full = true$, then EMERGENCY_WRITE($h_j$, $\mathrm{SM}_j$, $\emptyset$)

READ_4($X_1, \dots, X_\ell$)
        For $j = 0, 1, 2$
         (i) LOOKUP($X_1 \cup \cdots \cup X_\ell$, $GM_j$)
        (ii) execute (i) and (ii) of WRITE_4($X_1, \dots, X_\ell$) for $\ell = 0$
             where EMERGENCY_WRITE is replaced by EMERGENCY_READ,
             WRITE_M by D_READ_M, and $GM_j$ by $\widetilde{GM}$.
       (iii) D_READ_2(SM) (using $h_1$ and $h_2$)
        (iv) RESPOND REQUESTS
```

THEOREM 9.2. SIMULATION_4 *simulates an* $n\log\log(n)\log^*(n)$ *processor EREW PRAM on an n processor* DMM *using optimal delay* $O(\log\log(n)\log^*(n))$ *with high probability.*

PROOF. It is easy to see that the above algorithm is correct. For fixed j, loops (i) and (ii) of WRITE_4 run in time $O(\log\log(n)\log^*(n))$. For fixed j and r, loops (i) and (ii) follow the rules of the Extended_Process_2. Therefore, by Theorem 6.3 EMERGENCY is only invoked with inverse polynomial probability. Thus WRITE_4 (i) and (ii) a) run in time $O(\log\log(n)\log^*(n))$ with high probability. The analysis for READ_4 is analogous and yields the same performance as for WRITE_4. □

References

[1] N. Alon and J. H. Spencer. *The Probabilistic Method.* Wiley, New York, 1991.

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

[3] B. Bollobás. *Random Graphs.* Academic Press, London, 1985.

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

[5] B. S. Chlebus, K. Diks, T. Hagerup, and T. Radzik. Efficient simulations between concurrent-read concurrent-write PRAM models. *Proc. MFCS* '88, pages 231–239, 1988.

[6] B. S. Chlebus, K. Diks, T. Hagerup, and T. Radzik. New simulations between CRCW PRAMs. *Proc. MFCS* '89, pages 95–104, 1989.

[7] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic Perfect Hashing: Upper and Lower Bounds. Technical Report 77, Universität-GH Paderborn, FB Mathematik-Informatik, Jan. 1991. (Revised version of paper with same title that appeared in *Proc. 24th IEEE FOCS*, pages 524–531, 1988.) To appear in *SIAM J. Comput.*.

[8] M. Dietzfelbinger and F. Meyer auf der Heide. How to distribute a hash table in a complete network. *Proc. 22nd Ann. ACM Symp. on Theory of Computing*, pages 117–127, 1990.

[9] M. Dietzfelbinger and F. Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In M. S. Paterson, editor, *Proc. 17th ICALP*, pages 6–19, 1990. Lecture Notes in Computer Science, Vol. 443. Springer-Verlag, Berlin.

[10] F. E. Fich, P. L. Ragde, and A. Wigderson. Relations between concurrent-write models of parallel computation. *SIAM J. Comput.*, 17:606–627, June 1988.

[11] F. E. Fich, P. L. Ragde, and A. Wigderson. Simulations among concurrent-write PRAMs. *Algorithmica*, 3:43–51, 1988.

[12] J. Gil and Y. Matias. Fast hashing on a PRAM-designing by expectation. *Proc. SODA* '91, pages 271–280, 1991.

[13] J. Gil and Y. Matias. Leaders election without a conflict resolution rule—fast and efficient randomized simulations among CRCW PRAMs. *Proc. LATIN* '92, pages 204–218, 1992.

[14] J. Gil, Y. Matias, and U. Vishkin. Towards a theory of nearly constant time parallel algorithms. *Proc. FOCS* '91, pages 698–710, Oct. 1991.

[15] A. Karlin and E. Upfal. Parallel hashing—an efficient implementation of shared memory. *Proc. 18th Ann. ACM Symp. on Theory of Computing*, pages 160–168, 1986.

[16] C. P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. *Theoret. Comput. Sci.*, 71:95–132, 1990.

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

[18] C. McDiarmid. On the method of bounded differences. In J. Siemons, editor, *Surveys in Combinatorics*, pages 148–188. London Mathematical Society Lecture Note Series, Vol. 141. Cambridge University Press, Cambridge, 1989.

[19] K. Mehlhorn and U. Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Inform.*, 21:339–374, 1984.

[20] A. G. Ranade. How to emulate shared memory. *Proc. 28th IEEE Ann. Symp. on Foundations of Computer Science*, pages 185–194, 1987.

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

[22] E. Upfal. Efficient schemes for parallel communication. *J. Assoc. Comput. Mach.*, 31(3):507–517, 1984.

[23] L. G. Valiant. General purpose parallel architectures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity*, Chapter 18, pages 943–971. Elsevier, Amsterdam, 1990.