SIAM J. COMPUT.
Vol. 29, No. 5, pp. 1703–1739

CONTENTION RESOLUTION IN HASHING BASED SHARED MEMORY SIMULATIONS*

ARTUR CZUMAJ†, FRIEDHELM MEYER AUF DER HEIDE†, AND VOLKER STEMANN‡

Abstract. In this paper we study the problem of simulating shared memory on the distributed memory machine (DMM). Our approach uses multiple copies of shared memory cells, distributed among the memory modules of the DMM via universal hashing. The main aim is to design strategies that resolve contention at the memory modules. Extending results and methods from random graphs and very fast randomized algorithms, we present new simulation techniques that enable us to improve the previously best results exponentially. In particular, we show that an n-processor CRCW PRAM can be simulated by an n-processor DMM with delay $\mathcal{O}(\log\log\log n \log^* n)$, with high probability.

Next we describe a general technique that can be used to turn these simulations into time-processor optimal ones, in the case of EREW PRAMs to be simulated. We obtain a time-processor optimal simulation of an $(n\log\log\log n\log^* n)$-processor EREW PRAM on an n-processor DMM with delay $\mathcal{O}(\log\log\log n\log^* n)$, with high probability. When an $(n\log\log\log n\log^* n)$-processor CRCW PRAM is simulated, the delay is only by a $\log^* n$ factor larger.

We further demonstrate that the simulations presented can not be significantly improved using our techniques. We show an $\Omega(\log\log\log n/\log\log\log\log n)$ lower bound on the expected delay for a class of PRAM simulations, called topological simulations, that covers all previously known simulations as well as the simulations presented in the paper.

Key words. PRAM, distributed memory machine, randomized shared memory simulations, hashing

AMS subject classifications. 68Q05, 68Q10, 68Q25

PII. S009753979529564X

1. Introduction.

Parallel machines that communicate via a shared memory (*parallel random access machines, PRAMs*) are the most commonly used theoretical machine model for describing parallel algorithms (see, e.g., [16, 18, 28]). A PRAM consists of p processors $P_0,\ldots,P_{p-1}$, each having local memory, and a shared memory with cells $U=\{0,\ldots,m-1\}$. The processors work synchronously and have random access to the shared memory cells, each of which can store an integer. In this paper we deal only with *exclusive read exclusive write (EREW)* PRAMs and (ARBITRARY) *concurrent read concurrent write (CRCW)* PRAMs. On the EREW PRAM no pair of processors can simultaneously write to or read from the same memory location. On the (ARBITRARY) CRCW PRAM concurrent reading is allowed and, if several processors want to write to the same memory cell simultaneously, an arbitrary one of them succeeds. 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

*Received by the editors December 7, 1995; accepted for publication (in revised form) March 2, 1999; published electronically March 17, 2000. This work was supported by DFG-Graduiertenkolleg "Parallele Rechnernetzwerke in der Produktionstechnik," ME 872/4-1, by DFG-Sonderforschungsbereich 376 "Massive Parallelität: Algorithmen, Entwurfsmethoden, Anwendungen," by DFG Leibniz grant Me872/6-1, and by EU ESPRIT long term research project 20244 (ALCOM-IT).

http://www.siam.org/journals/sicomp/29-5/29564.html

†Heinz Nixdorf Institute and Department of Computer Science, University of Paderborn, D-33095 Paderborn, Germany (artur@uni-paderborn.de, fmadh@uni-paderborn.de).

‡Heinz Nixdorf Institute, University of Paderborn, D-33095 Paderborn, Germany. Current address: Deutsche Morgan Grenfell, Frankfurt, Germany.

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 theoretical model is the *distributed memory machine (DMM)*, in which the memory is divided into a limited number of memory modules, one module per processor. A DMM has n processors $Q_0, \ldots, Q_{n-1}$ which are connected by an interconnection network with a distributed memory consisting of n memory modules $M_0, \ldots, M_{n-1}$. In this paper we study DMMs with the *complete interconnection network* between processors and modules (cf. [7]). The computation of our DMM is synchronized. In a step each processor either performs a local computation or issues a read or write request for a memory cell x to the module holding x. Each module answers incoming requests. In this paper we shall focus on the DMM in which the modules obey the ARBITRARY conflict resolution rule: If more than one request is directed to a module, it serves an arbitrary one of them and ignores the others. However, the answer to the successful request is available to all processors accessing the module. It is easy to observe that an n-processor DMM can be simulated with constant delay on an n-processor ARBITRARY CRCW PRAM with $\Theta(n)$ shared memory cells and vice versa. Motivated by optical crossbar technology, there is another conflict resolution rule considered in the literature, the c-COLLISION rule. In this model a module can only answer if it gets at most c requests; otherwise a collision symbol is sent to all processors that wanted to access the module.

No matter which conflict resolution rule is used for a DMM, a module can respond to at most a constant number of accesses at a time. Thus DMMs exhibit the phenomenon of *memory contention*, in which an access request is delayed because of concurrent requests 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. Often the authors assumed that processors and modules are connected by a bounded degree network (e.g., by a mesh, a butterfly, or an expander), and packet routing is used to access the modules [17, 20, 21, 27, 31]. In this paper we consider DMMs with a complete interconnection between processors and modules, i.e., we focus on the issue of resolving memory contention.

All of our algorithms are randomized, and the time bounds hold *with high probability* (*w.h.p.*), i.e., with probability at least $1 - n^{-\alpha}$ for arbitrary constant $\alpha > 1$; the choice of α will affect the respective running times by at most a constant factor. We focus on simulations that minimize the *delay*, i.e., the time needed to simulate a parallel memory access of a PRAM on a DMM. Furthermore, we are interested in optimal simulations. We say a simulation of a p-processor PRAM on an n-processor DMM is *time-processor optimal* if the delay is $\mathcal{O}(\lceil p/n \rceil)$.

The most efficient simulations of shared memory are based on the idea of hashing, i.e., of distributing the shared memory cells of the PRAM (almost) randomly among the memory modules of the DMM. Mehlhorn and Vishkin [24] design a simple simulation of an n-processor EREW PRAM by an n-processor DMM with expected delay $\mathcal{O}(\log n / \log \log n)$, by storing each cell of the PRAM in a module that is determined by a single hash function. One can easily show that this result cannot be improved when each cell of the PRAM is represented only by one copy. Dietzfelbinger and Meyer auf der Heide [7] extend this result to the CRCW PRAM model and show that one step of an n-processor CRCW PRAM can be simulated by an n-processor DMM with expected delay $\mathcal{O}(\log n / \log \log n)$. Karp, Luby, and Meyer auf der Heide [19] break the $\log n / \log \log n$ bound by applying the idea of redundant storage represen-

tation, that is, by storing in the DMM more than one copy of each memory cell of the PRAM. In order to distribute some number of copies of each memory cell, randomly and independently chosen hash functions are used. Karp, Luby, and Meyer auf der Heide obtain a simulation of an n-processor CRCW PRAM on an n-processor DMM with delay $\mathcal{O}(\log\log n)$. They present how to get time-processor optimal simulations. These authors show that any simulation of one step of an n-processor EREW PRAM on an n-processor DMM with delay τ that uses only a constant number of hash functions can be turned into a time-processor optimal simulation with delay $\mathcal{O}(\tau \log^* n)$. In the case of CRCW PRAMs, the simulation is close to optimal: One step of an $(\tau \cdot n)$-processor CRCW PRAM can be simulated on an n-processor DMM with delay $\mathcal{O}(\tau \log^* n)$. Using the majority technique due to Upfal and Wigderson [32], Dietzfelbinger and Meyer auf der Heide [8] extend the $\mathcal{O}(\log\log n)$-delay simulation of Karp, Luby, and Meyer auf der Heide [19] to a much simpler schedule for an $\mathcal{O}(\log\log n)$-time simulation on the weaker c-COLLISION DMM, for some constant $c > 2$. Goldberg, Matias, and Rao [12] show that one can perform a time-processor optimal simulation with delay $\mathcal{O}(\log\log n)$, even on a 1-collision model (called also *optical communication parallel computer*, OCPC). Meyer auf der Heide, Scheideler, and Stemann [25] extend the algorithm from [8] and present a simulation on a DMM (with the ARBITRARY write conflict resolution rule) achieving delay $\mathcal{O}(\log\log n/\log\log\log n)$. This is the first simulation that beats the $\log\log n$ bound, however, it can not be turned into a time-processor optimal one because it uses nonconstant storage redundancy, that is, each memory cell of the PRAM has a nonconstant number of copies in the memory modules of the DMM. The techniques used in these papers do not seem to yield simulations with smaller delay. In particular, MacKenzie, Plaxton, and Rajaraman [22] and, independently, Meyer auf der Heide, Scheideler, and Stemann [25] show lower bounds for classes of algorithms that capture all these algorithms.

In this paper we design new shared memory simulations that improve all previously known results by an exponential decrease of the delay. The core of our simulations is a new analysis of a special sparse almost random graph, the access graph, which represents requests of the PRAM processors and an efficient use of log-star-time randomized algorithms, which leads to fast techniques for exploring neighborhoods of nodes in such sparse random graphs. The key steps of our simulations are partitioning techniques that make it possible to decompose every connected component of a random sparse graph into small pieces. Using different, more and more sophisticated partitioning techniques, we design simulations of one step of n-processor EREW PRAMs on n-processor DMMs with delay $\mathcal{O}(\log\log n/\log\log\log n)$, $\mathcal{O}(\sqrt{\log\log n}\log^* n)$, and finally $\mathcal{O}(\log\log\log n \log^* n)$, refining the simulation techniques step by step. Finally we transform all these bounds into simulations of one step of n-processor CRCW PRAMs on n-processor DMMs without asymptotic time loss.

Next we present a general technique that can be used to transform any simulation of an n-processor EREW PRAM on an n-processor DMM with delay $\tau \geq \log^* n$ that uses a constant number of hash functions into a time-processor optimal simulation. Our transformation relies on a new routing problem, called *all-but-linear routing*, which is a relaxed version of the $k - k$ relation routing problem. Using the ideas for $k - k$ relation routing developed for the OCPC model [1, 11], we show how to solve all-but-linear routing with random or almost random requests optimally for any $k = \Omega(\log^* n)$.

Finally we pinpoint the limit of our techniques. We analyze a *topological game* in graphs that is the essential part of the most efficient previously known simulations

[8, 12, 19, 22, 25], as well as the simulations presented in this paper. We show that a randomized topological game requires $\Omega(\log\log\log n/\log\log\log\log n)$ expected delay. This indicates that the techniques presented in the paper cannot lead to significantly better simulations than the ones presented in this paper.

Organization of the paper. We begin in section 2 with outlining techniques used by our simulations. Section 3 provides some basic tools. In section 4 we define the access graph and the access game and prove some properties about the distribution of sizes of connected components in a random graph that are essential for our proofs of the running time of the simulations. Section 5 contains the first simulation of an EREW PRAM, which has delay $\mathcal{O}(\log\log n/\log\log\log n)$, w.h.p. In section 6 we present two simulations of an EREW PRAM, one with delay $\mathcal{O}(\sqrt{\log\log n}\log^* n)$ and another with delay $\mathcal{O}(\log\log\log n\log^* n)$, w.h.p. Section 7 describes a transformation that turns EREW PRAM simulations into CRCW PRAM simulations. Section 8 analyzes the all-but-linear routing problem which is used in section 9 to get time-processor optimal simulations. In section 10 we present the lower bound for the topological game.

2. Outline of techniques. We start with the simulation of an n-processor EREW PRAM on an n-processor DMM. Our shared memory simulations are based on *redundant storage representation*; that is, we assume that the shared memory cells of the PRAM are distributed among the modules of the DMM using some number a of hash functions $h_1,\ldots,h_a : U \to \{0,\ldots,n-1\}$, so that copies of cell $u \in U$ are stored in the modules $M_{h_1(u)},\ldots,M_{h_a(u)}$. All such simulations assume that $h_1,\ldots,h_a$ are randomly chosen from a *high performance universal class* of hash functions as presented, e.g., by Siegel [29] or Karp, Luby, and Meyer auf der Heide [19]. A function randomly chosen from such a class of hash functions behaves almost like a random function but can be stored using little ($\mathcal{O}(\sqrt{n})$) space and can be evaluated in constant time. Actually, any $(2,\log^2 n)$-universal class of hash functions (cf. [6]) would be sufficient for our purposes. Upfal and Wigderson [32] observe how to use redundant storage representation in order to speed up shared memory simulations. They introduce the majority technique which utilizes the observation that accessing more than half of the a copies of a requested shared memory cell is sufficient for reading as well as for writing: If processor P wants to write to cell u, it updates more than half of the copies of u and adds a time-stamp indicating the PRAM time. If processor P wants to read cell u it reads at least half of the copies of u and takes the copy with the latest time-stamp.

Let us refer to the task of accessing b out of the a copies of each of the n requested shared memory cells as the *"b out of a" task*. A method for performing a "b out of a" task is called a *protocol*. For the analysis it is more convenient to consider a "1 out of c" task. It is possible to reduce a "b out of a" task to $\binom{a}{b-1}$ "1 out of $a-b+1$" tasks, each with a different subset of $a-b+1$ hash functions. For constant a this yields only a constant factor in the delay. In this paper we present shared memory simulations based on executing the "2 out of 3" task. In fact, because of the reasons mentioned above, we only analyze protocols for the "1 out of 2" task. From now on we will assume that each memory cell u of the PRAM is stored in the modules $M_{h_1(u)}$ and $M_{h_2(u)}$.

Our protocols for the "1 out of 2" task are based on the model introduced by Karp, Luby, and Meyer auf der Heide [19]. Let ε be a constant, $0 < \varepsilon < 1$, that will be specified later. Consider a batch of εn requests, for which the "1 out of 2" task has to be executed. For such a batch, we define the labeled *access graph* H as

follows. Its nodes correspond to the n modules of the DMM, and for each key u from the batch, H contains an edge labeled u between $M_{h_1(u)}$ and $M_{h_2(u)}$. Now a protocol for the "1 out of 2" task can be viewed as an *access game* on H. The access game is performed in rounds, and in each round each node of H (i.e., each module in the DMM) can remove one of its incident edges (i.e., one of the access requests directed to the module is processed). The access game is finished when all the edges have been removed from the graph.

2.1. Fast protocols for the "1 out of 2" task. The simple protocol for the "1 out of 2" task, as in [8, 12, 19, 25], corresponds to the access game in which, in each round, each node of H removes an arbitrary incident edge, i.e., processes one arbitrary request directed to it. MacKenzie, Plaxton, and Rajaraman [22] and Meyer auf der Heide, Scheideler, and Stemann [25] prove the tight lower bound of $\Omega(\log\log n)$ for the time needed by this protocol.

For the analysis of the access game or the protocol for the "1 out of 2" task, respectively, we also follow the idea of Karp, Luby, and Meyer auf der Heide [19], who analyze the structure of the access graph H. As h_1 and h_2 are almost random (i.e. randomly chosen from a high performance universal class), H is almost a random graph with n nodes and εn edges. In analogy to results on truly random graphs, these authors [19] show that H consists of connected components of size $\mathcal{O}(\log n)$, w.h.p., each of which is a tree with a constant number of additional edges, w.h.p. One can show (compare also [19]) that this property of H implies the existence of a schedule that works in constant time. The previously best algorithm to find this schedule takes $\mathcal{O}(\log\log n)$ time and yields the simulation from [8]. In contrast to formerly analyzed protocols for the access game, the heart of our simulations lies in finding fast protocols to execute the access game on H. In order to make the description of our protocols more intuitive, we assume that also the *module* may take part in the computation. Clearly this assumption is not critical, because we could let this work be carried out by the corresponding processor. Thus we assign the processors of the DMM both to the nodes and to the edges of H and allow the nodes (the modules) to decide which edges (if any) are to be removed. However, the knowledge about H is distributed among the processors at the beginning of the simulation, so that each processor knows just one edge. The only way the nodes may decide which edge to remove is to analyze their neighborhoods in H. There are two issues we have to deal with. First, each node has to explore its neighborhood quickly, and second, we have to find a rule to guide the nodes in their decision on which edge to remove on the basis of the structure of their neighborhood.

Exploring the neighborhood seems to be a complicated task. Even computing the degree of a node of H is complicated, because the edges in H are given in an arbitrary order and we do not have the adjacency list at hand. We show how each node v may explore certain properties of its *k-neighborhood*, i.e., the subgraph of H induced by the nodes at distance at most k from v, in time close to $\mathcal{O}(\log k)$. To succeed in this, we present a new precise analysis of the structure of H and apply extremely fast algorithms. We use probabilistic tools like the method of bounded differences [23] and a generalization of the Markov inequality to show that, w.h.p., for all i, the number of connected components of H of size at least i is bounded by $n/2^{bi}$ for some constant b.

Based on log-star techniques as developed in [3, 10, 14], we show how to use the structural properties of H mentioned above to compute certain properties of the k-neighborhood of all nodes of H in time $\mathcal{O}(\log k \log^* n)$. We then present a sequence of more and more involved protocols that use such information to design fast access

protocols.

We can extend all our results to the simulation of the CRCW PRAM. It is well known that by applying integer sorting to the memory requests issued in one step of the CRCW PRAM, one can reduce the simulation of one step of the CRCW PRAM to that of one step of the EREW PRAM. We observe that instead of integer sorting one can use an algorithm for *strong semisorting* [3], which can be solved on the DMM much faster than sorting. This allows us to turn any simulation of one step of an n-processor EREW PRAM by an n-processor DMM with delay $\tau \geq \log^* n$ into a simulation of one step of an n-processor CRCW PRAM on an n-processor DMM with delay $\mathcal{O}(\tau)$.

2.2. Time-processor optimal simulations. We present a general method for making simulations based on hashing multiple copies of the PRAM memory cells time-processor optimal. Consider a simulation of one step of an n-processor EREW PRAM on an n-processor DMM with delay not exceeding τ, w.h.p., that works by solving the "b out of a" task with $b > a/2$ (cf. the discussion in section 2). Further, suppose that it solves the "b out of a" task by running $\binom{a}{b-1}$ "1 out of $a-b+1$" tasks. We show that if a and b are constants, then the simulation can be made time-processor optimal with delay $\mathcal{O}(\tau)$ for $\tau \geq \log^* n$. This improves upon the $\mathcal{O}(\tau \log^* n)$ delay achieved in [19].

If we want to simulate one step of an $(n \cdot \tau)$-processor EREW PRAM on an n-processor DMM, each processor of the DMM simulates τ processors of the EREW PRAM. Hence, each processor of the DMM has a list of τ memory requests. We use $a+2$ hash functions and perform the simulation by solving the "$b+1$ out of $a+2$" task. By our discussion in section 2, one can solve the task by performing a "1 out of $a-b+2$" task for each subset of size $a-b+2$ of the $a+2$ hash functions. Therefore we focus on solving the "1 out of $a-b+2$" task. Thus let us suppose that we have $a-b+2$ hash functions. Our protocol consists of two phases. In the first phase, all but $\mathcal{O}(n)$ requests are satisfied in time $\mathcal{O}(\tau)$, using only one of the hash functions. We achieve this goal by analyzing a new routing problem, called all-but-linear routing. In the second phase we first evenly redistribute the remaining requests among the DMM processors such that each processor only gets a constant number of requests. Then we use the other $a-b+1$ hash functions and perform a protocol for the "1 out of $a-b+1$" task to satisfy the remaining requests. Since each processor has a constant number of requests to be sent, we may use the solution for the "1 out of $a-b+1$" task to perform this step in time $\mathcal{O}(\tau)$, w.h.p.

2.3. Lower bound for topological simulations. Our solutions for the access game, as well as all previously known solutions to the "1 out of 2" task, are special cases of a *topological game* on the access graph. In this game, in order to remove all the edges from the graph, each node first analyzes its neighborhood and then removes its incident edges in a way depending on the topology of the neighborhood. We prove an $\Omega(\log\log\log n/\log\log\log\log n)$ lower bound for the topological game, which indicates that our $\mathcal{O}(\log\log\log n \log^* n)$-delay PRAM simulation is almost optimal in the class of algorithms based on the topological game.

3. Preliminaries. This section describes basic techniques used in the paper. All our simulations are based on universal hashing, i.e., the shared memory cells are distributed among the memory modules using three or more hash functions, randomly drawn from a universal class of hash functions. The analysis of the simulations requires

high performance universal classes; i.e., if a function is chosen at random from the class, then its behavior will be close to that obtained by choosing a function at random from the class of all functions.

Since our simulations have $o(\log \log n)$ delay, we cannot afford to use deterministic and exact solutions for fundamental parallel problems, like sorting or prefix sums computation. Therefore, we use instead relaxed counterparts of these problems that can be solved by $\mathcal{O}(\log^* n)$-time randomized algorithms.

Section 3.1 briefly introduces the concept of hashing and universal hashing, section 3.2 states two tail estimates for dependent random variables, and section 3.3 presents results about fast randomized algorithms used in the paper.

3.1. Universal hashing. Our simulations require that the shared memory U of the PRAM is distributed among the memory modules of the DMM. For this we use hash functions that have properties similar to random functions. We use notation from Dietzfelbinger et al. [6], generalizing notions from Carter and Wegman [5].

For an integer k, define $[k] = \{0, \ldots, k-1\}$. Let $U = [m]$, where $m > n$.

DEFINITION 1 (see [6]). *A family $\mathcal{H}_{m,n}$ of hash functions mapping U into $[n]$ is (μ, k)-universal, if for each $u_1 < \cdots < u_k \in U$, $l_1, \ldots, l_k \in [n]$, and the hash function h drawn with uniform probability from $\mathcal{H}_{m,n}$, we have*

$$\Pr(h(u_1) = l_1, \ldots, h(u_k) = l_k) \leq \frac{\mu}{n^k} \ .$$

Such classes have been called sometimes μ strongly k universal or $((k)_\mu)$-independent. Notice that if $\mathcal{H}_{m,n}$ is (μ, k)-universal, then it is also (ν, j)-universal for all $\nu \geq \mu$ and $j \leq k$.

For our purposes we require a $(2, \log^2 n)$-universal class of hash functions $\mathcal{H}_{m,n}$, such that a random $h \in \mathcal{H}_{m,n}$ can be constructed quickly, stored using little space ($\mathcal{O}(\sqrt{n})$ suffices), and evaluated in constant time. For example, we can use an $(1, n^\varepsilon)$-universal class of hash functions (where ε is a small positive constant) described by Siegel [29] for the case $n = m$, with the extension to $(2, n^\varepsilon)$-universal class for arbitrarily large $m > n$ from [19]. For a detailed description of this class see [19].

3.2. Tail estimates. In this paper we mainly deal with dependent random variables. To bound the deviation of the sum of dependent random variables from the expected value, we use two different tail estimates. The first one is also called the *method of bounded differences*, given in this form by McDiarmid [23].

LEMMA 3.1 (the method of bounded differences). *Let $x_1, \ldots, x_n$ be independent random variables, where x_i takes values from a finite set A_i, for $i = 1, \ldots, n$. Suppose that the function $f : \Pi_{i=1}^n A_i \to \mathbb{R}$ satisfies $|f(\bar{x}) - f(\bar{x}')| \leq c_i$ whenever the vectors $\bar{x}$ and $\bar{x}'$ differ only in the ith coordinate. Let Y be the random variable $f(x_1, \ldots, x_n)$. Then, for any $t > 0$,*

$$\Pr(Y \geq E(Y) + t) \leq \exp\left(\frac{-2t^2}{\sum_{i=1}^n c_i^2}\right) \ .$$

Another tail estimate is a well-known generalization of the Markov inequality.

LEMMA 3.2 (the kth moment inequality). *Let Y be a random variable and $s > 0$. Then, for each $\alpha > 0$,*

$$\Pr(|Y| \geq \alpha) \leq \frac{E(|Y|^s)}{\alpha^s} \ .$$

3.3. Log-star algorithms. In this section we outline main algorithmic tools used by our algorithms. Some of them are subsumed by others. Nevertheless, we state all of them to make it easier to refer to specific tasks. If an array of size $2 \cdot n$ contains at least n objects, we will call the array *padded-consecutive*.

DEFINITION 2. *The* strong semisorting *problem is the following: Given n integers $x_1, x_2, \ldots, x_n$, $x_i \in [n]$, store them in a padded-consecutive array, such that all variables with the same value occur in a padded-consecutive subarray.*

DEFINITION 3. *The* all nearest one *problem is the following: Given n bits $x_1, x_2, \ldots, x_n$, find for each bit x_i the nearest 1's both to its left and to its right.*

DEFINITION 4. *The* approximate prefix sums *problem is the following: Given a sequence of nonnegative integers, $x_1, \ldots, x_n$, find a sequence $y_0 = 0$, $y_1, \ldots, y_n$, such that for $i \in [n]$, $y_i - y_{i-1} \geq x_i$, and*

$$\frac{1}{2}\sum_{j=1}^{i} x_j \leq y_i \leq 2\sum_{j=1}^{i} x_j.$$

The following lemma is a combination of several results.

LEMMA 3.3. *The following problems can be solved on the n-processor DMM in $\mathcal{O}(\log^* n)$-time with exponentially high probability, i.e., with probability $1 - 2^{-n^\varepsilon}$ for a constant $\varepsilon > 0$:*

(1) *strong semisorting,*

(2) *all nearest one, and*

(3) *approximate prefix sums.*

Proof. First, since an n-processor DMM can be simulated with a constant delay on an n-processor CRCW PRAM with $\Theta(n)$ shared memory, it is enough to prove the lemma for the n-processor CRCW PRAM that uses only $\mathcal{O}(n)$ space.

(1) Bast and Hagerup [3] show how to solve strong semisorting in $\mathcal{O}(\log^* n)$-time with the desired probability on a CRCW PRAM, provided the input is from $[n]$. For a general input, $\mathcal{O}(\log^* n)$-time perfect hashing [2, 10] reduces the problem to the solution of Bast and Hagerup. (See also [14, p. 275].)

(2) The all nearest one problem can be solved in $\mathcal{O}(\alpha(n)) = \mathcal{O}(\log^* n)$ time deterministically on a CRCW PRAM by an algorithm due to Ragde [26], and Berkman and Vishkin [4].

(3) Approximate prefix sums can be solved within the desired bound using an algorithm of Goodrich, Matias, and Vishkin [13]. □

4. The access graph. In this section we show how PRAM simulation can be modeled as the access game on the access graph. We also discuss basic properties of the access graph.

4.1. Definition and properties of the access graph. We start with the simulation of an EREW PRAM. The memory of the PRAM is hashed using three hash functions h_1, h_2, and h_3. That means each memory cell $u \in U$ of the PRAM is stored in the modules $M_{h_1(u)}, M_{h_2(u)}$, and $M_{h_3(u)}$ of the DMM. We call the representations of u in the $M_{h_i(u)}$'s the *copies* of u. We assume that all the hash functions used are drawn uniformly at random from a $(2, \log^2 n)$-universal class of hash functions $\mathcal{H}_{m,n}$ (see section 3.1). As mentioned in section 2, it is enough to analyze only protocols for the "1 out of 2" task, and therefore in our description we will use only two hash functions h_1 and h_2.

For technical reasons, we do not perform all n accesses to the shared memory simultaneously but split the requests into batches of size $n/2^{2c+6}$ for some constant $c \geq 1$ to be specified later. Since we only have a constant number of batches, this will slow down our algorithm only by a constant factor.

Let S denote such a batch. Let $G = ([n], E)$ be the labeled directed graph defined by two hash functions h_1, h_2 from $\mathcal{H}_{m,n}$ and the set of requests S. G has an edge $(h_1(u), h_2(u))$ labeled u for each $u \in S$. Note that parallel edges and self-loops are allowed in G, however all labels are distinct.

DEFINITION 5. *The* access graph *H is the labeled graph obtained from G by removing all directions from the edges. It consists of n nodes and $n/2^{2c+6}$ edges for some constant $c \geq 1$.*

The algorithms we present rely on the properties of the access graph. The following lemma is an extension of a result of Karp, Luby, and Meyer auf der Heide [19]. Define the size of a connected component C, denoted by $|C|$, to be the number of nodes it contains.

LEMMA 4.1. *For arbitrary positive constants l and c and for sufficiently large n*

(a) $\Pr($*H has a connected component of size at least $\frac{l}{c}\log n$*$) \leq n^{-l}$, and*

(b) *there is a constant $w \geq 1$ such that*

$$\Pr(H \text{ has a connected component } C \text{ with at least } |C| + w - 1 \text{ edges}) \leq n^{-l}.$$

Proof. For the proof of the lemma it suffices that h_1 and h_2 are chosen uniformly at random from any $(2, \log^2 n)$-universal class of hash functions. The proof of the lemma relies on the following claim.

CLAIM 4.2. *Let $k \geq 2, w \geq 0, k + w - 1 \leq \log^2 n$. The probability that there is a connected subgraph $G' \subseteq G$ such that G' contains k vertices and at least $k + w - 1$ edges is at most*

$$n^{-w+1} k^{w-2} 2^{-2c(k+w-1)} .$$

Proof. Let $\mathcal{G}_{k,w}$ be the set of all directed connected graphs on node set $[n]$ with k vertices and $k + w - 1$ edges labeled by elements of S. Since any connected subgraph $G' \subseteq G$ with k vertices and at least $k + w - 1$ edges must contain a subgraph from $\mathcal{G}_{k,w}$, it is sufficient to show that G contains a subgraph from $\mathcal{G}_{k,w}$ with probability at most $n^{-w+1} k^{w-2} 2^{-2c(k+w-1)}$.

An element of $\mathcal{G}_{k,w}$ can be generated by first choosing k vertices, then taking an undirected tree on these vertices and orienting the tree edges, then adding further w directed edges on the vertices of the tree, and finally assigning the labels to the chosen $k + w - 1$ edges. Therefore,[1]

$$\begin{aligned} |\mathcal{G}_{k,w}| &\leq \binom{n}{k} k^{k-2} 2^{k-1} k^{2w} 2^w \left(\frac{n}{2^{2c+6}}\right)^{k+w-1} \\ &\leq n^{2k+w-1} k^{2w-2} \left(\frac{e^2}{2^{2c+6}}\right)^{k+w-1} . \end{aligned}$$

For fixed $G' \in \mathcal{G}_{k,w}$ and randomly chosen h_1 and h_2, G' is a subgraph of G only if the directions and labels with respect to h_1 and h_2 coincide with G'. Since $k +$

[1] Throughout the paper the inequalities $k^k/k! < e^k$ and $(k/l)^l \leq \binom{k}{l} \leq (ek/l)^l$ will be used without further comment.

$w-1 \le \log^2 n$ and h_1, h_2 are independently chosen from a $(2, \log^2 n)$-universal class, the probability that G' is a subgraph of G is at most $\left(2/n^{k+w-1}\right)^2$. Therefore, the probability that there is some $G' \in \mathcal{G}_{k,w}$ that appears as a subgraph of G is at most

$$n^{-w+1}k^{2w-2}\left(\frac{4e^2}{2^{2c+6}}\right)^{k+w-1} \le n^{-w+1}k^{2w-2}2^{-2c(k+w-1)} \ . \qquad \square$$

To prove part (a) of the lemma we use Claim 4.2 with $w=0$ and $k=\frac{l}{c}\log n$. This yields an upper bound on the probability of the existence of a connected component of size at least $k=\frac{l}{c}\log n$.

To prove part (b) notice first that from part (a) we obtain that H has a connected component of size at least $\frac{l}{2c}\log n$ with probability at most $n^{-l/2}$. Observe also that for any constant s, H contains a vertex with at least s self-loops with probability at most n^{1-s}. Hence we may use Claim 4.2 to obtain the upper bound for the probability that H has a connected component C with at least $|C|+w-1$ edges:

$$\begin{aligned} n^{-l/2}+n^{1-w}+\sum_{k=2}^{\frac{l}{2c}\log n} n^{-w+1}k^{w-2}2^{-2c(k+w-1)} &\le n^{-l/2}+n^{1-w}+\sum_{k=2}^{\frac{l}{2c}\log n}(k/n)^{w-1} \\ &\le n^{-l/2}+n^{1-w}+n^{2-w} \ . \end{aligned}$$

Thus if we set $w=\lceil 2+l/2\rceil$, then the probability is at most n^{-l}. $\square$

LEMMA 4.3. *Let l, b, and c be any positive constants with $c-1 \ge b > 2$, $l > 2$, and let $1 < k \le \frac{1}{2b}\log n$. Then, for n large enough,*

$$\Pr\left(H \text{ has at least } \frac{n}{2^{bk}} \text{ connected components of size at least } k\right) \le n^{-(l-1)} \ .$$

Proof. We first consider connected components of H of size exactly k, and then we extend our analysis to connected components of size at least k.

Let $\mathcal{W}_k$ be the set of all subsets of size k of the set of n nodes. With each $W \in \mathcal{W}_k$ we associate a binary random variable $\mathcal{I}_W$, such that $\mathcal{I}_W = 1$ if the nodes from the set W form a connected component in H. Otherwise $\mathcal{I}_W = 0$.

We want to bound the random variable

$$X_{(k)} = \sum_{W \in \mathcal{W}_k} \mathcal{I}_W,$$

which is the number of connected components of size k. As the random variables $\mathcal{I}_{W_1}$ and $\mathcal{I}_{W_2}$ are not independent for $W_1, W_2 \in \mathcal{W}_k$, we use Lemma 3.2 to bound $X_{(k)}$. We compute the sth moment of $X_{(k)}$ with the following formula:

$$E(X_{(k)}^s) = E\left[\left(\sum_{W \in \mathcal{W}_k} \mathcal{I}_W\right)^s\right] = \sum_{(W_1,\ldots,W_s)\in(\mathcal{W}_k)^s} E(\mathcal{I}_{W_1}\mathcal{I}_{W_2}\cdots\mathcal{I}_{W_s}) \ .$$

Consider a fixed tuple $(W_1,\ldots,W_s)$. If there exists a pair of sets $W_i, W_j \in \{W_1,\ldots,W_s\}$ that are not disjoint and not equal, i.e., $W_i \ne W_j$ and $W_i \cap W_j \ne \emptyset$, then $E(\mathcal{I}_{W_1}\mathcal{I}_{W_2}\cdots\mathcal{I}_{W_s}) = 0$. This follows from the definition of the binary random variables because the two sets cannot both form a connected component of size exactly k. Hence, we only have to deal with the case that z sets, $1 \le z \le s$, are disjoint and

the other $s-z$ sets are equal to one of them. If we fix z, there are $\binom{s}{z}$ ways to choose the z sets that are disjoint and at most z^{s-z} ways to assign the remaining sets to one of them. Hence we can bound the sth moment in the following way:

$$E(X_{(k)}^s) \le \sum_{z=1}^{s} \binom{s}{z} z^{s-z} \sum_{\substack{(W_1,\ldots,W_z)\in(\mathcal{W}_k)^z \\ W_1,\ldots,W_z \text{ disjoint}}} E(\mathcal{I}_{W_1}\mathcal{I}_{W_2}\cdots\mathcal{I}_{W_z}) \ .$$

Fix z disjoint sets $W_1,\ldots,W_z$ and assume that $s(k-1) \le \log^2 n$. For every i, $1 \le i \le z$, let $\mathcal{T}_i$ be the set of all directed spanning trees on nodes of W_i with edges labeled by the elements of S. Notice that

$$E(\mathcal{I}_{W_1}\mathcal{I}_{W_2}\cdots\mathcal{I}_{W_z}) \le \sum_{T_1\in\mathcal{T}_1,\ldots,T_z\in\mathcal{T}_z} \Pr(\text{each of } T_1,\ldots,T_z \text{ is a subgraph of } H) \ .$$

Using arguments similar to those used in the proof of Claim 4.2, and by observing that the endpoints of the edges are chosen by hash functions from a $(2,\log^2 n)$-universal class, we obtain

$$E(\mathcal{I}_{W_1}\mathcal{I}_{W_2}\cdots\mathcal{I}_{W_z}) \le \left(k^{k-2}\cdot 2^{k-1}\cdot\left(\frac{n}{2^{2c+6}}\right)^{k-1}\right)^z \cdot \left(\frac{2}{n^{k-1}}\right)^{2z} \ .$$

Now we can bound $E(X_{(k)}^s)$:

$$\begin{aligned}
E(X_{(k)}^s) &\le \sum_{z=1}^{s}\binom{s}{z} z^{s-z}\binom{n}{k,k,\ldots,k,(n-zk)} k^{z(k-2)}\left(\frac{n}{2^{2c+5}}\right)^{z(k-1)}\left(\frac{4}{n^2}\right)^{z(k-1)} \\
&\le \sum_{z=1}^{s}\left(\frac{se}{z}\right)^z z^{s-z}\left(\frac{ne}{k}\right)^{zk} k^{z(k-2)}\left(\frac{1}{n2^{2c+3}}\right)^{z(k-1)} \\
&\le \sum_{z=1}^{s}\left(\frac{s}{z}\right)^z z^{s-z} n^z \left(\frac{e}{k^2}\cdot\frac{e^k}{2^{(2c+3)(k-1)}}\right)^z \\
&\le \sum_{z=1}^{s}\left(\frac{sn}{z^2 2^{ck}}\right)^z z^s \ .
\end{aligned}$$

Finally, Lemma 3.2 implies

$$\begin{aligned}
\Pr\left(X_{(k)} \ge \frac{n}{2^{bk+1}}\right) &\le \frac{E(X_{(k)}^s)}{\left(\frac{n}{2^{bk+1}}\right)^s} \\
&\le \sum_{z=1}^{s}\left(\frac{sn}{z^2 2^{ck}}\right)^z \cdot \left(\frac{z2^{bk+1}}{n}\right)^s \\
&= 2^s\cdot\sum_{z=1}^{s}\left(\frac{s}{z2^{(c-b)k}}\right)^z\cdot\left(\frac{z2^{bk}}{n}\right)^{s-z} \\
&\le 2^s\cdot\sum_{z=1}^{s}\left(\frac{s}{z2^k}\right)^z\cdot\left(\frac{z}{\sqrt{n}}\right)^{s-z} \\
&= 2^s\cdot\sum_{z=1}^{s}\left(\frac{s\sqrt{n}}{z^2 2^k}\right)^z\cdot\left(\frac{z}{\sqrt{n}}\right)^s \\
&\le 2^s\cdot s\cdot\left(\frac{1}{2}\right)^{ks} .
\end{aligned}$$

In the fourth inequality we need $c \geq b+1$ and $k \leq \frac{1}{2b}\log n$. By setting $s = (l+2)\log n/(k-1)$ we obtain

$$\Pr\left(X_{(k)} \geq \frac{n}{2^{bk+1}}\right) \leq n^{-(l+1)} \ .$$

Now we extend our analysis to connected components of size at least k. For this we have only to observe

$$\Pr\left(\sum_{t\geq k} X_{(t)} \geq \frac{n}{2^{bk}}\right) \leq \Pr\left(\sum_{t\geq k} X_{(t)} \geq \sum_{t=k}^{n} \frac{n}{2^{bt+1}}\right)$$
$$\leq \sum_{t\geq k} \Pr\left(X_{(t)} \geq \frac{n}{2^{bt+1}}\right) \leq n^{-l} \ . \qquad \square$$

Lemma 4.1 and Lemma 4.3 are the basis of the proof of the following lemma which is essential for the analysis of our simulations.

LEMMA 4.4. *Let H be the access graph with n nodes and $\frac{n}{2^{2c+6}}$ edges for some constant $c \geq 1$ defined as in Lemma 4.3. For all constants β and l such that $c \geq 2l(\beta+2)$, with probability at least $1-2(\frac{1}{n})^{l-1}$,*

$$\sum_{\substack{\text{connected components } C \\ |C|>1}} |C| \cdot 2^{|C|\,\beta} \leq n \ .$$

Proof. Let $b = \beta + 2$. We split the sum into two parts:

$$\sum_{1<|C|\leq \frac{1}{2b}\log n} |C| \cdot 2^{|C|\,\beta} + \sum_{\frac{1}{2b}\log n<|C|} |C| \cdot 2^{|C|\,\beta} \ .$$

Observe that because $c \geq 2bl$, we get $\frac{1}{2b}\log n \geq \frac{l}{c}\log n$. Therefore, the right-hand part is zero w.h.p. by part (a) of Lemma 4.1. Hence, by Lemma 4.3, with probability at least $1-2\left(\frac{1}{n}\right)^{l-1}$, the above sum is bounded by

$$\sum_{k=2}^{\frac{1}{2b}\log n} \frac{n}{2^{bk}} k 2^{\beta k} \leq n \sum_{k=2}^{\frac{1}{2b}\log n} \left(\frac{1}{2^{b-\beta-1}}\right)^{k} \leq n \ .$$

The last inequality holds as $b = \beta + 2$. $\square$

4.2. The access game. In the following we view a PRAM simulation, or more precisely a protocol for the "1 out of 2" task, as the following process on the access graph H, which we call the *access game* on H. Each processor that wants to access a shared memory cell $u \in U$ asks in each step either $M_{h_1(u)}$ or $M_{h_2(u)}$. Consequently, if a module M_j answers the request for cell u, then the edge labeled u is removed from H. That is, every node in H removes one incident edge (if any). The simulation ends when all the edges from H are removed.

Note that initially all the processors are assigned to the edges in H and that the nodes in H do not know the adjacent edges. We want to view the simulation from the point of view of the nodes; i.e., a node removes an incident edge. For this we assign to each node a processor that computes which of the incoming edges will be removed. Then it sends a message to the processor assigned to this edge to send a request to the respective module.

Remark 1. Lemma 4.1 immediately implies the $\mathcal{O}(\log\log n)$ implementation of the "1 out of 2" task due to Karp, Luby, and Meyer auf de Heide [19]. In each round, each module removes an arbitrary incident edge. Lemma 4.1 ensures that the connected components have at most $\mathcal{O}(\log n)$ edges. Thus, because in each round the number of edges in each connected component is at least halved, $\mathcal{O}(\log\log n)$ rounds suffice to remove all edges.

5. A simulation with delay $O(\log\log n/\log\log\log n)$. Throughout this section, H will denote the access graph, which is assumed to satisfy the conditions of Lemma 4.4. After the comments in section 4.2, the main challenge is how to remove the edges from the access graph. The first result is a randomized simulation of one step of an EREW PRAM with delay $\mathcal{O}(\log\log n/\log\log\log n)$, w.h.p. Its basic routine works with two hash functions and solves the "1 out of 2" task. The high level description of the protocol in terms of the processors and the modules is as follows. In each step, each module chooses among its incoming requests the one with the highest contention, i.e., for which the memory module storing the other copy gets most requests. For implementing the access protocol we use log-star techniques described in section 3.3 to get an $\mathcal{O}(\log^* n)$-time preprocessing algorithm for computing the number of requests directed to each module.

5.1. The neighbor-data-structure. We introduce a data structure called *neighbor-data-structure*. The neighbor-data-structure supports the following two operations on the access graph: (i) remove a set of edges and (ii) assign to each nonisolated node its neighbor with the maximum degree.

The preprocessing phase generates an array A of length $4n$. It contains, for each node v, a subarray A_v of length $\tilde{d}(v)$, $d(v) \le \tilde{d}(v) \le 2d(v)$. A_v contains the $d(v)$ edges incident to v and gaps in the remaining positions. Further, an array B of length n contains a pointer to the leftmost cell of A_v for each node v. In addition, the preprocessing phase assigns $\tilde{d}(v)2^{\tilde{d}(v)}$ processors to each nonisolated node v. By Lemma 4.4, at most n processors are needed for this assignment.

For a subgraph H' of H, the neighbor-data-structure is derived from the one for H by simply removing the missing edges from the subarrays A_v. The assignment of processors and the lengths of the A_v's remain unchanged.

NDS Preprocessing.

- *Represent each (undirected) edge (i,j) by two ordered pairs $[i,j]$ and $[j,i]$.*
- *Perform strong semisorting with respect to the first coordinate.*
- *Compute for each node v the size $\tilde{d}(v)$ of the subarray A_v containing all edges adjacent to v.*
- *Allocate $\tilde{d}(v)\,2^{\tilde{d}(v)}$ processors to each node v.*

After strong semisorting, the second step of the preprocessing, all edges (the corresponding ordered pairs) adjacent to a node v of degree $d(v)$ appear in a subarray A_v of size $\tilde{d}(v) = \mathcal{O}(d(v))$. The algorithm for the all nearest one problem can be used to find the first and the last edge of each node and therefore to compute the size $\tilde{d}(v)$ of the subarray A_v. The following lemma bounds the time needed for the preprocessing.

Lemma 5.1. NDS Preprocessing *builds up the neighbor-data-structure and can be performed in $\mathcal{O}(\log^* n)$ time on an n-processor DMM, w.h.p.*

Proof. The time bound follows from Lemma 3.3. Note that $d(v) \le \tilde{d}(v) \le 2d(v)$. Hence, by Lemma 4.4, the total size of the neighbor-data-structure is n. In addition, since the degree of v is bounded by the size of its connected component, w.h.p.,

the total number of allocated processors to each connected component is at most $2\,|C|^2\,2^{2\,|C|}$. Therefore, Lemma 4.4 ensures that we allocate only a linear number of processors, w.h.p. □

The following lemma shows how to dynamically manipulate the neighbor-data-structure.

LEMMA 5.2. *Once the neighbor-data-structure is built, one can update the neighbor-data-structure after deletion of any set of edges in constant time on an n-processor DMM.*

Proof. Since each edge e knows its positions in the arrays A_v, one can remove any edge by changing the corresponding positions into gaps. □

The next lemma describes the functionality of the neighbor-data-structure.

LEMMA 5.3. *Given the neighbor-data-structure for some subgraph of H, on an n-processor DMM, it is possible in constant time to assign to each nonisolated node a neighbor of maximum degree.*

Proof. Let us look at the array A_v as an array containing marked objects (corresponding to edges adjacent to v) and unmarked objects (corresponding to gaps). To compute the degree of v it is sufficient to compute the number of marked objects in A_v. Since there are only $2^{\tilde{d}(v)}$ combinations of marked and unmarked objects in A_v, one can compute this number in constant time, using $\tilde{d}(v)$ processors for each combination. Note that this amount of processors is available for each node because we allocated an exponential number of processors to each node. Now each node v can get the degrees of all its neighbors in constant time with $\tilde{d}(v)$ processors. To find a node with the maximum degree one can use the standard maximum finding algorithm that runs (deterministically) in constant time with $(\tilde{d}(v))^2$ processors [16, p. 72]. □

5.2. Schedule and analysis of the simulation. In section 4 we have shown that if every node always removes an arbitrary adjacent edge, then a simulation with delay $\Theta(\log\log n)$ results. We want to describe a more efficient protocol, SIMULATION 1, using the neighbor-data-structure.

SIMULATION 1.

- NDS PREPROCESSING*: Build up the neighbor-data-structure.*
- *Repeat until no more edges are left:*
  - *In parallel, each node removes the edge pointing to a neighbor with highest degree.*
  - *Update the neighbor-data-structure.*

We bound the number of iterations of the algorithm.

LEMMA 5.4. SIMULATION 1 *is finished after $\mathcal{O}(\frac{\log\log n}{\log\log\log n})$ iterations, w.h.p.*

Proof. We say an access graph H *survives* k iterations of SIMULATION 1, if at least one edge is left after performing k iterations. We want to construct structures that are embeddable in the access graph H, if it survives k iterations of SIMULATION 1. Let $k \geq 1$. A *k-fork* consists of k different nodes (called *leaves*) connected to one node (called the *center*) that is connected to another node called the *parent node* of the k-fork. A *k-witness* is defined recursively as follows. A 1-witness is a path of length 5 (i.e., with 6 nodes); the two nodes of degree one are the *leaves*. For $k > 1$, a k-witness is a $(k-1)$-witness each leaf of which is a parent node of a different k-fork. All leaves of the k-forks are called the leaves of the k-witness. Note that a k-witness has $2k!$ leaves. An example of a 4-witness is given in Figure 1. The proof of the lemma is based on the following claim.

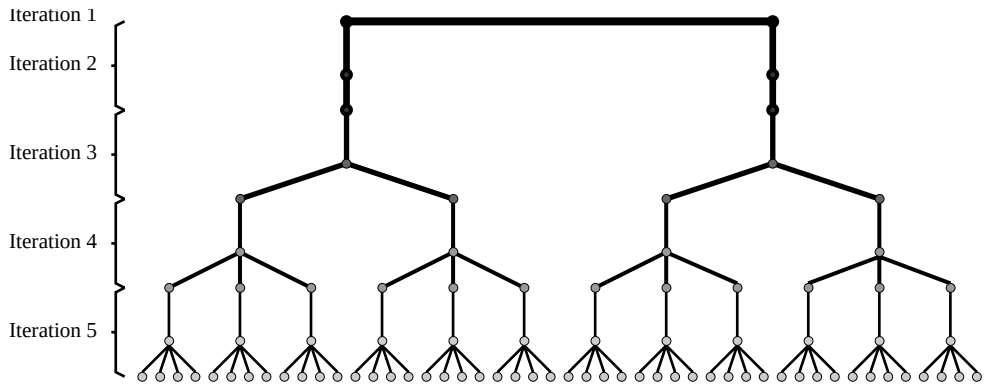

FIG. 1. *A 4-witness.*

CLAIM 5.5. *If H survives k iterations of* SIMULATION 1*, then there is an embedding of a k-witness W in H that maps edges that have a node in common to different edges of H.*

Proof. The proof of the claim is by induction on k. If H survives one iteration, then a path of length 5 must be embeddable in H. Assume now that the claim holds for k and let H survive $k+1$ iterations.

Perform the first iteration on the graph H. A subgraph $\tilde{H} \subseteq H$ which survives k iterations will be left. From the induction hypothesis, we know that a k-witness $\tilde{W}$ can be embedded in $\tilde{H}$ in the way described in the claim. For this structure to survive the last iteration we extend it to a witness-structure W that must be embeddable in H. Since in the first iteration no edge from $\tilde{H}$ has been removed, each node from $\tilde{H}$ must have removed an incident edge in $H - \tilde{H}$. As $\tilde{W}$ is a k-witness, it consists of a $(k-1)$-witness each leaf of which is a parent node of a k-fork. Because each center of a k-fork removes an edge from $H - \tilde{H}$ that points to a neighbor with the maximum degree, its degree in H is at least $k+2$. Thus every leaf v in $\tilde{W}$ must have a neighbor u (different than the center) of degree at least $k+2$. Hence v must be the parent node of a $(k+1)$-fork that consists of v, u and $k+1$ neighbors of u other than v and the center of the k-fork. This defines the required embedding of W in H. □

Assume that SIMULATION 1 needs more than k iterations on the graph H. Then by Claim 5.5 a k-witness W is embedded in a connected component C of H.

By Lemma 4.1, $|C| = \mathcal{O}(\log n)$, and C consists of a tree and a constant number ζ of additional edges, w.h.p. In the worst case, at each node of C, at most ζ edges from W can be embedded into these ζ edges of C. Therefore, there must exist a $(k-\zeta)$-witness $\tilde{W}$ that can be embedded one-to-one in C. As $|\tilde{W}| \geq 2(k-\zeta)!$, by the definition of a $(k-\zeta)$-witness, the embedding of W in C as described in Claim 5.5 uses $\Omega((k-\zeta)!)$ nodes of C. Since $|C| = \mathcal{O}(\log n)$, we can embed only a k-witness W for $k = \mathcal{O}(\frac{\log \log n}{\log \log \log n})$. □

This lemma together with Lemmas 5.1, 5.2, and 5.3 implies the following theorem.

THEOREM 5.6. *One can simulate one step of an n-processor EREW PRAM on an n-processor DMM with delay $\mathcal{O}(\log \log n / \log \log \log n)$, w.h.p.*

6. Faster simulations. In this section we extend the ideas from the previous section. We present an algorithm that not only computes information on the 2-neighborhood (degrees of neighbors) but explores a larger neighborhood of each node. Then, a second algorithm is presented that removes all edges from the access graph

in constant time, assuming each node knows its whole connected component. As a first consequence we give a simple protocol that significantly improves the delay of the simulations presented before. Finally, we present the $\mathcal{O}(\log\log\log n \log^* n)$ delay simulation.

6.1. The path-access-structure. In this subsection we develop some algorithmic utilities for almost random graphs, especially methods for exploring a large neighborhood of a node, which is essential for our algorithms. Throughout this section, H will denote the access graph which fulfills the conditions of Lemma 4.4. The *k-neighborhood* of a node v is the subgraph of H containing the nodes and the edges that are reachable from v by a path of length at most k.

Our simulations use a data structure, called *path-access-structure* (PAS), that allows fast access to all simple paths of length at most k in H. We prove a simple lemma to bound the total length of all simple paths in H.

LEMMA 6.1. *Let C be a connected component that is a tree with a constant number ζ of additional edges and v be a node in C. Then the total number of simple paths starting at v is $\mathcal{O}(|C|)$.*

Proof. The total number of simple paths that do not use the additional edges is $|C|$. The other simple paths can use each additional edge only once. Therefore, the number of simple paths using additional edges can be bounded by $\zeta! 2^{\zeta+1}|C| = \mathcal{O}(|C|)$. □

Note that, by Lemma 4.1, each connected component C of H has the properties stated in Lemma 6.1, w.h.p. As each simple path in C has length at most $|C|$, we can bound the total length of all simple paths in C by $\mathcal{O}(|C|^3)$, w.h.p. Hence, by the statement made in Lemma 4.4 about the distribution of the sizes of these components, the total length of all the simple paths is $\mathcal{O}(n)$, w.h.p.

The *path-access-structure for (simple paths of length at most) k* is a data structure that supports the following operations on H: (i) remove a set of edges, (ii) assign to each nonisolated node v the node with minimum identifier in the k-neighborhood of v, (iii) assign to each nonisolated node in H an incident edge that begins a simple path of length at least k (if there exists such an edge), and (iv) for all edges e and indices r, $1 \le r \le k$, give the number of simple paths of length exactly r that start with edge e.

Now we describe the implementation details of the path-access-structure for k. We store the simple paths of length at most k in an array S of length $\mathcal{O}(n)$, which consists of padded-consecutive subarrays S_v, one for each node v of H. Each S_v contains a representation of all paths starting in v, each one in consecutive cells. In order to access the paths we build up an array P which consists of padded-consecutive subarrays P_v, one for each node v of H. P_v contains a pointer to the header of each path starting at node v together with the length of this path.

We first give a high level description of how to apply the doubling technique to build up the path-access-structure for simple paths of length at most k.

PAS PREPROCESSING.

All nodes are active.

Initialize all the P_v and S_v as empty.

For $r = 0$ to $\log k$ perform the following iteration for all active nodes v in parallel:

- *Using P_v, compute an approximate (within a factor of 2) number $\tilde{\delta}_r(v)$ of simple paths of length at most 2^r starting at v.*
- *Using S_v, compute an approximate (within a factor of 2) total length $\tilde{\gamma}_r(v)$ of all simple paths starting at v.*

- *Update the sizes of S_v and P_v using $\tilde{\delta}_r$ and $\tilde{\gamma}_r$.*
- *Using the doubling technique, find all simple paths of length l, $2^r < l \leq 2^{r+1}$, starting at v and store them in S_v and P_v.*
- *Inactivate all nodes that are not beginnings of a simple path of length 2^{r+1}.*

LEMMA 6.2. *In time $\mathcal{O}(\log k \log^* n)$, with linear total work, the path-access-structure for all simple paths in H of length at most k can be built on an n-processor DMM, w.h.p., using the procedure* PAS PREPROCESSING.

Proof. The algorithm is based on the standard doubling technique (see, e.g., [16]). We perform $\log k + 1$ iterations and ensure the following invariant after r iterations, $0 \leq r \leq \log k$: Each node v has already found all simple paths of length at most 2^r that start at v, stored them in a padded-consecutive subarray S_v and stored the pointers to each path together with its length in a padded-consecutive subarray P_v. If a given node has already found all its simple paths, then it is *inactive*. Otherwise it is *active*. Note that in each iteration of the algorithm only active nodes participate.

When $r = 0$, then all the paths of length 1 are exactly the edges incident to v. Thus, this 0th iteration of PAS PREPROCESSING is covered by the construction of the neighbor-data-structure. The NDS PREPROCESSING and Lemma 5.1 can be used to build up the neighbor-data-structure. Within $\mathcal{O}(\log^* n)$ time and linear work, one can move all the edges incident to v to a padded-consecutive subarray at the beginning of A_v, w.h.p. (compare section 3.3). This creates the adjacency list of v. Thus, we can create the arrays S_v and P_v. Additionally, we inactivate all isolated nodes.

We perform the $(r+1)$st iteration, for $r \geq 0$, using the doubling technique. Let v be any active node and C_v be its connected component. Note that since v is active, we have $|C_v| \geq 2^r$. First, v computes how many simple paths of length at most 2^r start at v. Because it is hard to compute this value exactly, each node v computes a value $\tilde{\delta}_r(v)$ which is not smaller and at most 2 times larger than the number of paths that start at v. Since the subarray P_v containing the pointers to all simple paths starting at v is padded-consecutive, simply finding the first and the last such paths makes it possible to compute $\tilde{\delta}_r(v)$ after performing the algorithm for the all nearest one problem. Now we compute an approximation $\tilde{\gamma}_r(v)$ of the total length of all simple paths stored at S_v. Since the lengths of all such paths are stored at P_v, we compute $\tilde{\gamma}_r(v)$ using the approximate prefix sums algorithm for all v. The difference between the last cumulative path-lengths in two consecutive subarrays P_v and $P_{v'}$ gives us $\tilde{\gamma}_r(v)$.

Let (x, y) be the last edge of any simple path p of length 2^r which starts at v. To find all paths of length l, $2^r < l \leq 2^{r+1}$, starting with p, we must combine p with all paths of length at most 2^r starting at y. Then we remove the anomalies, i.e., the paths that create cycles.

Observe that, using the values $\tilde{\delta}_r(y)$ and $\tilde{\gamma}_r(y)$, we know how big the new arrays P_v and S_v have to be (for each path p, P_v has to be extended by $\tilde{\delta}_r(y)$ and S_v by $2^r \cdot \tilde{\delta}_r(y) + \tilde{\gamma}_r(y)$). This space allocation can be done using global approximate prefix sums. Observe that $\tilde{\delta}_r(v) = \mathcal{O}(|C_v|)$ and $\tilde{\gamma}_r(v) = \mathcal{O}(|C_v| \cdot 2^r) = \mathcal{O}(|C_v|^2)$, w.h.p., using Lemma 6.1. Hence the size of the new P_v is $\mathcal{O}(|C_v|^2)$, and the size of the new S_v is $\mathcal{O}(|C_v|^3)$, w.h.p.

It is easy to compute the length of each newly created path to maintain P_v. To update S_v we have only to copy the old paths from S_v and concatenate the paths from v to y with simple paths from y. Hence these operations can be performed in constant time with total work proportional to the sizes of the new P_v and S_v. This means that the total work for all nodes is $\sum_{|C| \geq 2^r} \mathcal{O}(|C|^4)$, and the total running

time in each iteration is $\mathcal{O}(\log^* n)$, since it is dominated by the cost of computing $\tilde{\gamma}_r(v)$, $\tilde{\delta}_r(y)$ and of allocating the subarrays P_v and S_v.

Finally, we have to remove those obtained paths that are not simple. We identify each path in S_v with the position of the first node. Then we perform strong semisorting within all arrays S_v with respect to the pairs [path, a node on the path]. Now, if there is more than one pair $[p, y]$, which can be verified by applying the all nearest one algorithm, then the path p is not simple and we eliminate it. Strong semisorting within all arrays P_v can be used to remove nonsimple paths from these arrays. Using the lengths of the paths in the P_v's, approximate prefix sums makes it possible to remove all nonsimple paths in the arrays S_v. Hence we can maintain the P_v's and S_v's to be padded-consecutive. Now we inactivate a node v if the new P_v contains no path of length 2^{r+1}.

The running time of iteration r is $\mathcal{O}(\log^* n)$ with $\mathcal{O}(\sum_{|C|\geq 2^r} |C|^4)$ total work. Therefore, the total work of the algorithm is

$$\mathcal{O}\left(\sum_{r=0}^{\log k} \sum_{|C|\geq 2^r} |C|^4\right) = \mathcal{O}\left(\sum_{C} |C|^4 \log |C|\right),$$

and by Lemma 4.4, this is $\mathcal{O}(n)$. □

Now we show how to maintain the path-access-structure dynamically, that is, when we allow removing edges from the graph. The proof of the following lemma is in the spirit of the proof of Lemma 6.2.

LEMMA 6.3. *Assume that the path-access-structure has already been built. Then, after removing some edges from the graph, it can be updated in time $\mathcal{O}(\log^* n)$ on an n-processor DMM, w.h.p.*

Proof. First we perform the preprocessing step and run strong semisorting on all arrays S_v with respect to the names of the edges. (Of course, it causes no problem to find and look at the edges instead of the nodes.) The result is an array $\mathcal{R}$ of linear size, such that for each edge e in a padded-consecutive subarray of $\mathcal{R}$ all the occurrences on all simple paths together with the pointers to these paths in P_v's are stored.

Now, when an edge e is removed, we can easily remove from the arrays P_v and S_v all simple paths on which e occurs. To remove all the edges, one needs constant time and work proportional to the size of $\mathcal{R}$. If one requires that all the subarrays P_v and S_v still have to be padded-consecutive, then additional $\mathcal{O}(\log^* n)$ time is needed. □

LEMMA 6.4. *Assume that the path-access-structure for k has been built. Then, to each nonisolated node one can assign the node with the minimum identifier in its k-neighborhood in $\mathcal{O}(\log^* n)$ time on an n-processor DMM.*

Proof. All simple paths that start at v and are of length at most k are stored in a padded-consecutive subarray S_v. Thus, this subarray contains exactly the nodes from the k-neighborhood of v. Now each node v can find the node w with the minimal identifier in its k-neighborhood in $\mathcal{O}(\log^* n)$ time. Each node v has $|S_v| = \mathcal{O}(|C_v|^2)$ candidates, where C_v denotes the connected component v belongs to, and using $|S_v|^2 = \mathcal{O}(|C_v|^4)$ processors it can find the node with minimum identifier in constant time [16, p. 72]. $\mathcal{O}(\log^* n)$ time is needed for assigning processors to the nodes by using approximate prefix sums. □

LEMMA 6.5. *Assume that the path-access-structure for k is given. One can assign to each node its incident edge that begins a simple path of length at least k in constant time on an n-processor DMM.*

Proof. For each node v the array P_v immediately gives such an edge. □

The next lemma shows how to use the path-access-structure for k to count the number of all simple paths of length at most k.

LEMMA 6.6. *Suppose that the path-access-structure for k is given. Then, for all edges e and integers r, $1 \le r \le k$, the number of simple paths of length exactly r that start with edge e can be computed in $\mathcal{O}(\log^* n)$ on an n-processor DMM, w.h.p.*

Proof. For each simple path in all S_v's we consider the pairs [starting edge of the path, length of the path]. We perform strong semisorting with respect to these keys. In this way, all the simple paths (in fact their representatives) that start with the same edge e and are of the same length r are stored in a padded-consecutive subarray, which we call $X_{e,r}$. If $y_{e,r}$ denotes the number of such paths, then $y_{e,r} \le |X_{e,r}| \le 2y_{e,r}$. We allocate $|X_{e,r}| \cdot 2^{|X_{e,r}|}$ processors to the pair $[e, r]$ and compute $y_{e,r}$ in the same way as in the proof of Lemma 5.1 in constant time. Now we have to show that we use only n processors. Let C_e be the connected component e belongs to. By Lemma 4.1, $y_{e,r} = \alpha|C_v|$ and each connected component C has at most $\beta|C|$ edges, for some constants α and β. Hence

$$\sum_{e\in E}\sum_{r=0}^{k} |X_{e,r}|\cdot 2^{|X_{e,r}|} \le \sum_{C}\sum_{e\in C} |C|\cdot(2y_{e,r})2^{2y_{e,r}} \le \sum_{|C|>1} 2\alpha\beta|C|^3 2^{2\alpha|C|} \le \sum_{|C|>1} |C|2^{b|C|}$$

for some constant b. Lemma 4.4 ensures that this is bounded by n. □

A *k-branch* of a node v in the access graph H is the set of all different simple paths of length at most k that start with the same edge incident to v. Clearly, for every node v the number of its k-branches equals its degree. Define LEVEL(r) of a k-branch of a node v to be the set of all simple paths of length r, $0 < r \le k$, of this k-branch. The *weight* of a k-branch of a node v is the bit-vector $\overline{w} = (w_1, \ldots, w_k)$, where w_r is 1 if and only if the number of simple paths in LEVEL(r) of the branch is at least 2^{r-1}. We order the weights with respect to the lexicographical ordering. Informally, a k-branch is lexicographically larger than another k-branch, if it is more similar to a complete binary tree with respect to the number of nodes in each level.

LEMMA 6.7. *Suppose that the path-access-structure for k is given. Then one can assign to each node in H an incident edge that begins a k-branch with maximum weight in $\mathcal{O}(\log^* n)$ time on an n-processor DMM.*

Proof. Using Lemma 6.6, each node can get the weights of all incident k-branches in $\mathcal{O}(\log^* n)$ time. Let C_v denote the connected component containing node v. Each node has $\mathcal{O}(|C_v|)$ k-branches, each of depth $\mathcal{O}(|C_v|)$. Hence it can find a k-branch with the maximum weight in constant time using $|C_v|^4$ processors using the standard algorithm (see, e.g., [16, p. 72]). Hence all nodes together can find the required k-branches in constant time on a p-processor DMM with $p = \sum_v |C_v|^4 = \sum_{|C|>1} |C|^5$. By Lemma 4.4, $p \le n$. □

6.2. Cleaning up connected components. The $\mathcal{O}(\log\log n/\log\log\log n)$-time simulation from section 5 is based on very local information. Each node looks only at its neighbors in the access graph and, based on their degrees, chooses one incident edge. The main idea leading to improvements of that result is to analyze the access graph in a nonlocal way and to try to use information on as many nodes and edges as possible.

The notion of the k-neighborhood plays the crucial role. Instead of looking only at the neighbors, now each node v will base the decision which incident edge to remove on the structure of its k-neighborhood. We will explore the k-neighborhood of each

node in H. As we have seen in Lemma 6.2, essentially all information on the k-neighborhood can be computed in time $\mathcal{O}(\log k \log^* n)$. In this subsection we will describe a process that removes all edges from a connected component with diameter δ in time $\mathcal{O}(\log \delta \log^* n)$.

CLEANING UP CONNECTED COMPONENTS.

- PAS PREPROCESSING*: Build up the path-access-structure.*
- *Perform for each node in parallel:*
  - *Each node v obtains the node w with the minimal identifier in its component.*
  - *If $v \neq w$, then v finds all nodes $u_1, u_2, \ldots, u_r$, such that (v, u_i) is the first edge of a simple path from v to w.*
  - *v removes the edges (v, u_i), $1 \leq i \leq r$.*

The access protocols described in previous papers (e.g., in [19], see also Remark 1 in section 4.2) show how to remove all edges of a connected component C of H in time $\mathcal{O}(\log(|C|))$. The following lemma proves that CLEANING UP CONNECTED COMPONENTS achieves time $\mathcal{O}(\log(\text{diameter of } C))$. Note that this does not give fast simulations on its own, because H has a connected component of diameter $\Omega(\log n / \log\log n)$, with constant probability (see Lemma 10.2).

LEMMA 6.8. *Let δ be the maximum diameter of the connected components of H. Then* CLEANING UP CONNECTED COMPONENTS *removes all edges in H in time $\mathcal{O}(\log \delta \log^* n)$, w.h.p.*

Proof. As it is shown in Lemma 6.2, one can build up the path-access-structure for δ in time $\mathcal{O}(\log \delta \log^* n)$ on an n-processor DMM. Because for each node its δ-neighborhood is equal to its connected component, Lemma 6.4 allows us to find the nodes w in CLEANING UP CONNECTED COMPONENTS in $\mathcal{O}(\log^* n)$ time.

By Lemma 4.1 we have $r = \mathcal{O}(1)$, w.h.p. Now we may use the array S from the path-access-structure to find for each node v the nodes $u_1, \ldots, u_r$ in $\mathcal{O}(1)$ time. Notice also that each edge e of H must be of the form (v, u_i) for some v and i. Therefore, at the end of the algorithm all the edges are removed. Since $r = \mathcal{O}(1)$ w.h.p., the last step of the algorithm takes $\mathcal{O}(1)$ time, w.h.p. ∎

6.3. An $O(\sqrt{\log\log n}\log^* n)$-delay simulation. The $\mathcal{O}(\sqrt{\log\log n}\log^* n)$ simulation consists of three phases and may be described on a high level in the following way. Let $k = 2^{\sqrt{\log\log n}}$.

SIMULATION 2.

- PAS PREPROCESSING: *Build up the path-access-structure for k.*
- *Repeat $2 \cdot \log\log n / \log k$ times:*
  - *Each node removes an incident edge which is the beginning of a simple path of length at least k, if there exists such an edge.*
  - *Update the path-access-structure.*
- CLEANING UP CONNECTED COMPONENTS.

Since the clean-up phase removes all edges from H, the correctness of the algorithm is obvious. We show that after performing the loop in the second phase of SIMULATION 2, we have partitioned H so that the diameter of each connected component is at most $2k$. Therefore, the clean-up phase can be performed in time $\mathcal{O}(\log k \log^* n)$ on an n-processor DMM, w.h.p., as shown in Lemma 6.8.

We say that a connected component C of H *survives* t iterations of SIMULATION 2 if at least one edge of C is not removed at the end of the tth iteration in the second phase. The idea of the proof is to bound the size of C.

LEMMA 6.9. *After the second phase of* SIMULATION 2*, the diameter of each connected component is at most* $2k$*, w.h.p.*

Proof. Consider a connected component C of H that survives t iterations and any maximal connected subgraph C_t of C of size larger than 1 that is left after t iterations. Define inductively a sequence $C_t, C_{t-1}, \ldots, C_0$ such that for every i, $0 \leq i < t$, C_i is a maximal connected subgraph of C that survives the ith iteration and contains C_{i+1}.

If the diameter of C_t is greater than $2k$, then every node v on C_t is the beginning of a simple path of length k. Hence, according to the elimination rule of the first phase of SIMULATION 2, every node has removed in this iteration an incident edge not belonging to C_t which was the beginning of a simple path of length at least k. (Note that if the diameter of C_t is greater than $2k$, then this also holds for C_{t-i}, $1 \leq i \leq t$.)

The $|C_t| \cdot k$ edges of these paths are edges of the access graph H. Since each path is simple, edges of one path are embedded injectively. From Lemma 4.1 we know that each connected component C consists of $|C| + \mathcal{O}(1)$ edges. Hence, it has only a constant number ζ of cycles. If we embed edges of two of these paths to the same edge in H, then the sum of the two paths contains a cycle in H. Therefore, at least $|C_t| - \zeta$ of these simple paths of length k are disjoint from each other in the embedding, that is, C_{t-1} has $k \cdot (|C_t| - \zeta) \geq \frac{k}{2} \cdot |C_t|$ edges.

This recursion shows: if a connected component C survives t iterations in the second phase of SIMULATION 2, and if a connected part of C left at the end of the tth iteration has diameter greater than $2k$, then C must be of size $\Omega(k^t/2^t)$. Now assume that this is the case for $t = 2 \cdot \log\log n / \log k$. Then C must be of size $\Omega(\log^2 n)$, which contradicts Lemma 4.1. □

From Lemmas 6.2, 6.3, and 6.5, it follows that it is possible for each node to analyze in the first phase its k-neighborhood in time $\mathcal{O}(\log k \log^* n)$ and each iteration in the second phase can be executed in time $\mathcal{O}(\log^* n)$, w.h.p. Therefore, if we set $k = 2^{\sqrt{\log\log n}}$ and apply the majority technique we obtain the following theorem.

THEOREM 6.10. *One can simulate one step of an n-processor EREW PRAM on an n-processor DMM with $\mathcal{O}(\sqrt{\log\log n}\log^* n)$ delay, w.h.p.*

6.4. An $O(\log\log\log n \log^* n)$-delay simulation. In this subsection we describe a simulation of an n-processor EREW-PRAM on an n-processor DMM that improves all previously known simulations exponentially. Essentially, we show how to partition each connected component using information about the $(\log\log n)$-neighborhood so that the diameter of the largest connected component is $\mathcal{O}((\log\log n)^2)$. Then we can apply the CLEANING UP CONNECTED COMPONENTS procedure to remove the remaining edges in H. To achieve an improvement compared to SIMULATION 2, we use the knowledge about the k-neighborhood in a more efficient way.

SIMULATION 3.

- PAS PREPROCESSING*: Build up the path-access-structure.*
- *Each node v removes the incident edge which is the beginning of a k-branch with maximal weight.*
- CLEANING UP CONNECTED COMPONENTS.

Lemmas 6.2 and 6.7 make sure that the first two phases of SIMULATION 3 can be performed in time $\mathcal{O}(\log k \log^* n)$, w.h.p. Now we prove that at the beginning of the clean-up phase the maximum diameter of each connected component in H is at most $\mathcal{O}(k^2)$, w.h.p., for $k \geq \log\log n$.

LEMMA 6.11. *Let $k \geq \log\log n$, and ζ denote the number of cycles in the access graph H. After the second phase of* SIMULATION 3*, w.h.p., H does not contain any simple path of length more than $(\zeta + 1) \cdot (2k + 1)^2$.*

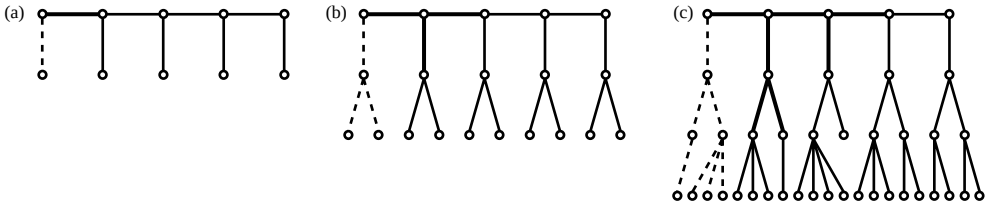

FIG. 2. (a) *the path $\tilde{p}$ with the edges that are removed by the nodes of $\tilde{p}$;* (b) *the path $\tilde{p}$ with the edges required by the elimination rule on* LEVEL(2)*;* (c) *the path $\tilde{p}$ with the edges required by the elimination rule on* LEVEL(3).

Proof. Assume, to the contrary, that at the beginning of the clean-up phase a simple path p of length $(\zeta+1)\cdot(2k+1)^2$ survives. Let us call a branch that starts with an edge from p the *path-branch*, and let us call any k-branch chosen in the second phase of SIMULATION 3 by a node from p the *side-branch* of this node. Since no edge of the path p has been removed in the first step, for each node of p, the side-branch differs from the path-branch. Lemma 4.1 ensures that $\zeta = \mathcal{O}(1)$. Hence there exists a subpath $\tilde{p} = (v_0, v_1, \ldots, v_{2k})$ of p, such that all vertices of the side-branches of nodes from $\tilde{p}$ are not contained in any cycle of length smaller than $2k$.

If the weight of a k-branch satisfies $w_1 = w_2 = \cdots = w_r = 1$, then we call it *$r$-complete*. We show that for each node from $\tilde{p}$ the side-branch must be r-complete; for all nodes v_i, $0 \le i \le 2k - r$, the right path-branch (starting from the edge (v_i, v_{i+1})) is r-complete; and for all nodes v_i, $r \le i \le 2k$, the left path-branch (starting from the edge (v_i, v_{i-1})) is also r-complete.

We prove the desired properties by induction on the levels (defined in section 6.1). For an example see Figure 2.

LEVEL(1). Because no edge of a simple path $\tilde{p}$ of length $2k$ was removed in the first step of the algorithm, each node from $\tilde{p}$ had to remove an incident edge not belonging to $\tilde{p}$. Therefore for each node from $\tilde{p}$ the side-branch is 1-complete. Similarly, since for each node from $\tilde{p}$ each path-branch has $w_1 = 1$, for each node v_i, $0 \le i \le 2k-1$, the right path-branch is 1-complete, and for each node v_i, $1 \le i \le 2k$, the left path-branch is 1-complete.

LEVEL(r). Now assume that $r > 1$ and for each node of $\tilde{p}$ the side-branch and the respective path-branches are $(r-1)$-complete. Consider a node v_i, $0 \le i \le 2k - r$, and the edge (v_i, v_{i+1}). Since the side-branch and the right path-branch of v_{i+1} are both $(r-1)$-complete and disjoint and have no cycle of length smaller than or equal to k, the right path-branch of v_i also must be r-complete. The nodes v_i, $r \le i \le 2k$, can be treated in a similar way.

This implies that we need a connected structure containing at least $2k \cdot 2^{k-1}$ nodes in H for a simple path of length $(\zeta+1)\cdot(2k+1)^2$ to survive the second phase of SIMULATION 3. For $k \ge \log\log n$ this contradicts Lemma 4.1. ∎

This yields the following result.

THEOREM 6.12. *One can simulate one step of an n-processor EREW PRAM on an n-processor DMM in $\mathcal{O}(\log\log\log n \log^* n)$ time, w.h.p.*

7. Reduction from CRCW PRAM to EREW PRAM. In this section we show how to reduce the problem of simulating a CRCW PRAM on a DMM to the simulation of an EREW PRAM. It is standard to build simulations of CRCW PRAMs by combining sorting with simulations for EREW PRAMs. However, since we are interested in very fast simulations for which sorting is too slow, our reduction is

based on strong semisorting. Suppose that processor P_i of the CRCW PRAM wants to access memory cell $u_i \in U$, for $1 \leq i \leq n$. In order to reduce the problem to the EREW PRAM we have to show how to deal with duplicate requests to the same memory cell. We consider only reading accesses; writing accesses can be performed similarly.

We first perform strong semisorting on pairs $(u_1, 1), \ldots, (u_n, n)$ with respect to the first coordinate. This results in the addresses with the same value being stored in a padded-consecutive subsequence of the output sequence. Using the algorithm for the all nearest one problem each u_i can test whether it is the first element in the padded-consecutive subsequence or not. If so, we call u_i a *leader*. Notice that if u_i is not a leader, then the solution to the all nearest one problem provides the pointer to its leader. Now we perform an EREW simulation on the DMM with requests only from leaders. Afterwards all leaders have accessed their cell u_i and can broadcast the value to the duplicates in constant time.

LEMMA 7.1. *If one step of an n-processor EREW PRAM can be simulated on an n-processor DMM with delay t, w.h.p., then one step of an n-processor CRCW PRAM can be simulated on an n-processor DMM with delay $\mathcal{O}(t + \log^* n)$, w.h.p.*

Coupled with Theorem 6.12, this yields the following theorem.

THEOREM 7.2. *One step of an n-processor CRCW PRAM can be simulated on an n-processor DMM with $\mathcal{O}(\log^* n \log\log\log n)$ delay, w.h.p.*

8. All-but-linear routing. In this section we analyze a relaxed version of the routing problem, the *all-but-linear routing* problem. Unlike in the general routing problem, we do not require that all messages are delivered to their destinations, but we have to route a large fraction of the messages. The motivation for our study is the problem of transforming simulations of n-processor PRAMs by n-processor DMMs into time-processor optimal simulations. As we will show in section 9, a fast solution for all-but-linear routing can be directly applied to obtain efficient time-processor optimal PRAM simulations.

DEFINITION 6. *In the* all-but-linear routing *problem on an n-processor DMM each processor Q_i has k keys $u_{i,1}, u_{i,2}, \ldots, u_{i,k}$, each to be sent to a destination from $M_0, \ldots, M_{n-1}$. The task is to deliver all but $\mathcal{O}(n)$ of the $n \cdot k$ keys.*

In our application we will need the destinations of the keys to be chosen almost randomly. The actual input for the PRAM simulation consists of kn *distinct* integers $x_{1,1}, \ldots, x_{n,k}$ from $[m]$, and we let $u_{i,j} = h(x_{i,j})$, for $i \in [n], j \in [k]$, where h is chosen uniformly at random from a $(2, \log^2 n)$-universal class $\mathcal{H}_{m,n}$ of hash functions (see section 3.1). We call such an instance of all-but-linear routing *quasi-random*. In fact, we could give a simpler solution if h were a truly random function (see [30]).

Our main result in this section is the following theorem.

THEOREM 8.1. *The quasi-random all-but-linear routing problem can be solved in time $\mathcal{O}(k + \log^* n)$, w.h.p., for $k \leq \sqrt[4]{\log n}$.*

Our solution for quasi-random all-but-linear routing is based on an algorithm for the $k - k$ relation routing problem by Goldberg et al. [11].

DEFINITION 7. *In the $k - k$ relation routing problem, each processor wants to send at most k messages to other processors which are assumed to be distinct. The destinations can be arbitrary except that each processor is the destination of at most k messages.*

8.1. Reduction to $k - k$ relation routing. We will reduce quasi-random all-but-linear routing to $k-k$ relation routing by achieving the following two goals. First,

we will ensure that each processor has only keys with distinct destinations. Second, we will restrict the problem to processors that are destinations of $\mathcal{O}(k)$ messages.

The first problem can be solved using an element distinctness algorithm for each processor. That is, each processor Q_i sequentially runs through its keys $u_{i,1}, \ldots, u_{i,k}$ and finds all duplicate destinations. This can be done by using, for example, bucket sorting, in $\mathcal{O}(k)$ time and with $\mathcal{O}(n)$ space for each processor. Let us call a processor *bad* if it has keys that have the same destination.

LEMMA 8.2. *If $k = \mathcal{O}(\sqrt[4]{\log n})$, then $\mathcal{O}(\log^2 n)$ processors are bad, w.h.p.*

Proof. Let X_i, $i = 1, \ldots, n$, denote the binary random variable indicating if processor P_i is bad or not. Processor P_i is bad with probability at most $\binom{k}{2}\frac{2}{n} \le \frac{k^2}{n}$, that is, $P(X_i = 1) \le k^2/n$. Let $X = \sum X_i$. Since the keys have destinations chosen by a function h taken at random from a $(2, \log^2 n)$-universal class of hash functions, the choices of any $ek^2 \log n$ processors are almost independent (cf. Definition 1). In particular, we can give the following bound for $X = \sum_{i=1}^{n} X_i$:

$$\Pr(X \ge ek^2 \log n) \le \binom{n}{ek^2 \log n} 2 \left(\frac{k^2}{n}\right)^{ek^2 \log n} \le 2 \left(\frac{1}{\log n}\right)^{ek^2 \log n} \le \frac{1}{n^{\log \log n}}. \quad \square$$

Hence, for all-but-linear routing, we can leave all keys of bad processors unprocessed, subsuming them in the $\mathcal{O}(n)$ remaining keys. To reduce the remaining all-but-linear routing to $k - k$ relation routing, we need the following lemma that bounds the number of keys that have their destination at modules with load at least $\Omega(k)$.

LEMMA 8.3. *Let h be chosen uniformly at random from a $(2, \log^2 n)$-universal class of hash functions with range $[n]$, and $1 \le k \le \frac{\log n}{\log \log n}$. If $n \cdot k$ keys are distributed among n locations using h, then at most $\frac{n}{2^k}$ keys are in locations with load at least $8k$, w.h.p.*

Proof. Denote the nk keys by $b_0, \ldots, b_{nk-1}$. A location is *heavy* if its load is at least $8k$. We associate with each key b_i a binary random variable $\mathcal{E}_i$ with $\mathcal{E}_i = 1$ if b_i is in a heavy location; otherwise $\mathcal{E}_i = 0$.

We want to bound the random variable $X = \sum_{i=0}^{kn-1} \mathcal{E}_i$. Again we use the kth moment inequality (Lemma 3.2) and therefore compute $E(X^s)$ for $s = \Theta(\log n)$.

$$E(X^s) = \sum_{(j_1,\ldots,j_s)\in[kn]^s} E(\mathcal{E}_{j_1} \cdots \mathcal{E}_{j_s}) = \sum_{z=1}^{s} \sum_{\substack{(j_1,\ldots,j_s)\in[kn]^s \\ z \text{ of the } j_i \text{ are different}}} E(\mathcal{E}_{j_1} \cdots \mathcal{E}_{j_s}) \ .$$

First we bound each term $E(\mathcal{E}_{j_1} \cdots \mathcal{E}_{j_s})$. Fix keys $(j_1, \ldots, j_s) \in [kn]^s$ such that z of the j_i are different. $E(\mathcal{E}_{j_1} \cdots \mathcal{E}_{j_s})$ is equal to the probability that each of the z keys has its destination in a heavy location. This probability is bounded by the sum over all possible locations of the z keys of the probabilities that the z keys have destinations in given locations and that the locations are heavy. Since h is chosen from a $(2, \log^2 n)$-universal class of hash functions and $z < \log^2 n$, the probability that the z keys have destinations in given r locations, $1 \le r \le z$, is bounded by $r^z \cdot 2/n^z$. For $r \le z/8k$, we trivially estimate the probability that r locations are heavy by 1. If $z/8k < r \le z$, then we observe that in order to r locations be heavy there must be other $8kr - z$ keys with destinations in the r locations. Therefore in that case we bound the probability that the r locations are heavy by $\binom{nk}{8kr-z} \cdot 2 \cdot (r/n)^{8kr-z}$.

Hence, we obtain

$$\begin{aligned}
E(\mathcal{E}_{j_1}\cdots\mathcal{E}_{j_s}) &\le \sum_{1\le r\le \frac{z}{8k}} \binom{n}{r} 2\left(\frac{r}{n}\right)^z + \sum_{\frac{z}{8k}<r\le z} \binom{n}{r} 2\left(\frac{r}{n}\right)^z \binom{nk}{8kr-z} 2\left(\frac{r}{n}\right)^{8kr-z} \\
&\le 2\sum_{r\le \frac{z}{8k}} \left(\frac{ne}{r}\right)^r \left(\frac{r}{n}\right)^z + 4\sum_{\frac{z}{8k}<r\le z} \left(\frac{ne}{r}\right)^r \left(\frac{r}{n}\right)^z \left(\frac{nker}{(8kr-z)n}\right)^{8kr-z} \\
&\le 4\left(\frac{8ekn}{z}\right)^{\frac{z}{8k}} \left(\frac{z}{8kn}\right)^z + 8\left(\frac{en}{z}\right)^z \left(\frac{z}{n}\right)^z \left(\frac{kez}{(8k-1)z}\right)^{(8k-1)z} \\
&\le 4\left(\frac{z}{\sqrt{n}}\right)^z + \left(\frac{1}{2}\right)^{8(k-1)z} \\
&\le 5\left(\frac{1}{2}\right)^{8(k-1)z}.
\end{aligned}$$

The third inequality holds for $k = \mathcal{O}(\log n/\log\log n)$ and $z = o(n)$, which holds as $z \le s = \Theta(\log n)$. In this case it is easily seen that the terms in each sum at least double when r increases. Hence we can bound each sum by two times the largest term. Now we can bound $E(X^s)$:

$$\begin{aligned}
E(X^s) &\le \sum_{z=1}^{s} \sum_{\substack{(j_1,\ldots,j_s)\in[kn]^s \\ z \text{ of the } j_i \text{ are different}}} 5\left(\frac{1}{2}\right)^{8(k-1)z} \\
&\le \sum_{z=1}^{s} \binom{s}{z} (kn)^z z^{s-z} 5\left(\frac{1}{2}\right)^{8(k-1)z} \\
&\le 5\sum_{z=1}^{s} z^s \left(\frac{sekn}{z^2 2^{8(k-1)}}\right)^z \\
&\le 10 s^s \left(\frac{ekn}{s2^{8(k-1)}}\right)^s \\
&\le \left(\frac{n}{2^{2k}}\right)^s.
\end{aligned}$$

Again, each term of the sum in the third line at least doubles for large enough n if z increases. Hence, we have bounded the sum by two times the largest term.

As the $\mathcal{E}_i$ and therefore also X are nonnegative random variables, we finally get

$$\Pr(X \ge \alpha) \le \frac{E(X^s)}{\alpha^s} \le \left(\frac{n}{2^{2k}\alpha}\right)^s.$$

For $\alpha = \frac{n}{2^k}$ we get

$$\Pr\left(X \ge \frac{n}{2^k}\right) \le \left(\frac{1}{2^k}\right)^s.$$

For any positive l, if $s \ge l\log n$, then this probability is bounded by n^{-l}. □

8.2. $k-k$ relation routing. It remains to show that we can perform the $k-k$ relation routing in time $\mathcal{O}(k)$. As was observed by Anderson and Miller [1], one can solve the off-line version of the problem in k steps. However, we are interested in this

problem when each processor has only the information about the messages it has to send, and it can learn about the messages of the other processors only by sending and receiving messages.

It is easy to see that any $k-k$ relation routing can be performed in time k^2 on a DMM, but our aim is to do this faster. Anderson and Miller [1] considered the $k-k$ relation routing problem on a model equivalent to a 1-collision DMM and showed how to solve $\log n - \log n$ relation routing in $\mathcal{O}(\log n)$ expected time. Valiant [33] considered the more general problem. He described a $\Theta(\log n + k)$ expected time algorithm on the same model. This result was improved by Goldberg et al. [11], to an algorithm for the $k-k$ relation routing problem running in $\mathcal{O}(\log\log n + k)$ time, w.h.p. We describe a faster algorithm on an DMM.

THEOREM 8.4. *The $k-k$ relation routing problem can be solved on an n-processor DMM in $\mathcal{O}(\min\{\log^* n + k, k^2\})$ communication steps for any $k \le n^{1/11}$.*

The proof of this theorem uses known methods and techniques from [1, 11]. In fact it is very similar in spirit to the results presented there. Nevertheless we present the full proof in order to make our description of the all-but-linear routing algorithm self-contained.

To prove the theorem we first describe the randomized routing protocol. It consists of $\mathcal{O}(\log k)$ rounds. In round i the problem of a $k/2^{i-1} - k/2^{i-1}$ relation routing will be reduced to a $k/2^i - k/2^i$ relation routing problem. After the last round all but n/k of the $n \cdot k$ keys will be delivered. Then we redistribute the remaining keys among the processors and deliver them in $\mathcal{O}(k)$ steps, w.h.p.

Before presenting more details we introduce some notation. We call a processor *overloaded* at the beginning of round i if the number of keys it has not already delivered is greater than $k/2^{i-1}$. Similarly we call a module *overloaded* at the beginning of phase i if the number of keys that it still has to receive is greater than $k/2^{i-1}$. A module or a processor, respectively, becomes overloaded in round i if it was not overloaded at the beginning of round i but it is at the beginning of round $i+1$. The idea of the algorithm is that in each round only nonoverloaded processors *participate*, i.e., only those keys participate that are not sent by an overloaded processor. Additionally we consider only keys sent to a nonoverloaded module. The main problem is to bound the number of overloaded processors and modules and hence the number of keys that are left after the $\log k$ rounds. Let c be a positive constant, $c \ge 80$.

$k-k$ RELATION ROUTING.

(1) *For $1 \le i \le \log(\frac{k}{5\log k}) + 1$, perform the following round i:*
Only processors that have fewer than $k/2^{i-1}$ keys left to send participate.
- *Repeat $c \cdot k/2^i$ times:*
Each participating processor chooses a random number r from $[\frac{k}{2^{i-1}}]$ and tries to send the rth undelivered message, if it exists.

(2) *All processors that have fewer than $5\log k/2$ nondelivered keys send the keys one by one, each key $5\log k/2$ times.*

(3) *Redistribute evenly the remaining keys using approximate parallel prefix sums applied to the numbers of nondelivered keys belonging to each processor.*

(4) *Deliver the remaining keys.*

The proof of the theorem is based on the following lemma which bounds the number of keys not participating.

LEMMA 8.5. *Let $k \le n^{1/11}$, and let i be an arbitrary round in the first step of the $k-k$ relation routing algorithm above. Then at most n/k^4 modules and at most n/k^4 processors become overloaded in round i, w.h.p.*

Proof. Consider an arbitrary round i of the algorithm, $1 \leq i \leq (\frac{k}{5 \log k}) + 1$. Let x_j denote the sequence of integers randomly chosen by processor P_j during round i.

We first prove that the number of overloaded modules at the beginning of round $i+1$ can be bounded by n/k^4, w.h.p. Let Y_M be the number of modules that become overloaded during round i. Let M be a module that is not overloaded at the beginning of round i, and let m_j denote the number of participating keys that are destined for M at the jth step of round i. Notice that M cannot become overloaded in round i if m_j is less than $k/2^i$. Therefore, in that case, the probability that M does not receive a message in the jth step is

$$\left(1 - \frac{1}{k/2^{i-1}}\right)^{m_j} \leq \left(1 - \frac{1}{k/2^{i-1}}\right)^{k/2^i} \leq e^{-1/2} \ .$$

Observe now that if M is overloaded at the end of round i, then in at least $ck/2^i - k/2^i$ steps of this round no message is received by M. Therefore, we can bound the probability that M is overloaded at the end of round i by

$$\binom{c \cdot k/2^i}{k/2^i} \cdot (e^{-1/2})^{c \cdot k/2^i - k/2^i} \leq (ec)^{k/2^i} \cdot (e^{-1/2})^{(c-1) \cdot k/2^i} = \left(c \cdot e^{3/2-c/2}\right)^{k/2^i} \ .$$

Since we have assumed that $i \leq \log(\frac{k}{5 \log k}) + 1$ and $c \geq 80$, we may bound the probability by $1/2k^4$. This immediately implies that $E(Y_M)$, the expected number of modules that become overloaded in round i, is at most $n/2k^4$. Now we apply the method of bounded differences (Lemma 3.1) to bound the probability that $Y_M \geq n/k^4$. For this we view Y_M as the image of a function f_M:

$$Y_M = f_M(\{x_j | P_j \text{ is participating at the beginning of round } i\}) \ .$$

Observe that any change of the choice of P_j in a single step from key α to β may change only the load of the two modules being destinations of α and β. Since all choices of P_j are independent, a change of the value of x_j for any j may change the number of modules overloaded in round i by at most $2ck/2^i \leq ck$. Hence we apply Lemma 3.1 to get

$$\Pr(Y_M \geq n/k^4) \leq \Pr(Y_M \geq n/2k^4 + E(Y_M)) \leq \exp\left(-\frac{2(n/k^4)}{n(ck)^2}\right) = \exp\left(-\frac{2n}{c^2 k^{10}}\right).$$

This probability is exponentially small for $k \leq n^{1/11}$.

Now we want to show that, w.h.p., at most n/k^4 processors become overloaded during round i. Let Y_P be the number of processors that become overloaded during round i. Let P be any participating processor in round i and let p_j denote the number of keys that P still has to send in the jth step of round i.

Let $d_{l,j}$ denote the number of participating keys in the jth step that have the same destination as the lth key that P has to send. We consider only those P that do not send to overloaded modules. Therefore each $d_{l,j}$ is less than or equal to $k/2^{i-1}$. Note that P cannot become overloaded in round i if p_j is ever less than $k/2^i$. Thus we consider only the case $p_j \geq k/2^i$ for all j. In that case the probability that P sends a key successfully in the jth step is at least

$$\sum_{l=1}^{p_j} 2^{i-1}/k \cdot \left(1 - 2^{i-1}/k\right)^{d_{l,j}-1} \geq \sum_{l=1}^{p_j} 2^{i-1}/k \cdot \left(1 - 2^{i-1}/k\right)^{k/2^{i-1}} \geq 1/2e^2 \ .$$

Therefore, using arguments similar to those above, we obtain that the probability that P stops participating in round i is at most

$$\binom{ck/2^i}{k/2^i} \cdot (1-1/2e^2)^{(c-1)k/2^i} \leq \left(e \cdot c \cdot (1-1/2e^2)^{c-1}\right)^{k/2^i} .$$

Since $i \leq \log(\frac{k}{5\log k}) + 1$ and we set $c \geq 80$, this probability is bounded by $1/2k^4$. As in the proof of the first part, we can conclude that the expected number of processors that become overloaded during round i is at most $\frac{n}{2k^4}$. One can verify that if the value of x_j changes for any j, then Y_P changes by at most ck. Therefore, by the method of bounded differences (Lemma 3.1), the probability that Y_P is greater than n/k^4 is at most

$$\Pr\left(Y_P \geq \frac{n}{k^4}\right) \leq \Pr\left(Y_P \geq \frac{n}{2k^4} + E(Y_P)\right) \leq \exp\left(-\frac{2(n/k^4)^2}{n \cdot (ck)^2}\right) = \exp\left(-\frac{2n}{c^2 k^{10}}\right) .$$

This probability is exponentially small for $k \leq n^{1/11}$. □

Now we proceed with the proof of Theorem 8.4.

Proof of Theorem 8.4. After the first step of $k-k$ Relation Routing a key may be left if either its processor has fewer than $5\log k/2$ nondelivered keys, or its processor is overloaded in some round and stops participating, or the module where the key should be sent is overloaded in some round. If the key is in the processor with fewer than $5\log k/2$ nondelivered keys, then after sending it $5\log k/2$ times in the second step either it will be delivered to the destination, or there are more than $5\log k/2$ keys sending to the destination module. The latter means that the module is overloaded at some round in the first step. Therefore, after the second step, we are left only with such keys for which either the processor to which they belong or the corresponding destination module is overloaded at some round in the first step. Because an overloaded processor affects at most k keys and an overloaded module affects at most k^2 keys, the number of keys that are left at the end of the $\log k$ rounds can be bounded using Lemma 8.5 by

$$\log k \left(k^2 \frac{n}{k^4} + k\frac{n}{k^4}\right) \leq \frac{n}{k} .$$

Using approximate parallel prefix computation (Lemma 3.3) applied to the number of nondelivered keys belonging to each processor, we can redistribute the n/k remaining keys among the modules in $\mathcal{O}(\log^* n)$ time, such that each processor gets at most a constant number, w.h.p. Finally, each processor can now deliver its keys in $\mathcal{O}(k)$ steps.

Hence, altogether we need $\mathcal{O}(\log^2 k + k + \log^* n)$ time. □

At the beginning of $k-k$ relation routing a module is called *overloaded* if it gets $\omega(k)$ requests.

Lemma 8.3 bounds the number of keys affected by the modules getting $\omega(k)$ requests by $kn/2^k$. These keys and the keys from the bad processors have to be added to the other keys that will not be processed during the $\mathcal{O}(\log k)$ rounds of $k-k$ relation routing. Therefore, the total number of keys not processed during the $\mathcal{O}(\log k)$ rounds of $k-k$ relation routing remains linear. This finishes the description of quasi-random all-but-linear routing and completes the proof of Theorem 8.1.

9. Time-processor optimal EREW PRAM simulations. In this section we want to present time-processor optimal simulations of EREW PRAMs on DMMs,

that is, simulations of $(n \cdot t)$-processor EREW PRAMs on n-processor DMMs with delay not exceeding $\mathcal{O}(t)$, w.h.p. For this we first consider the problem of accessing 1 out of c copies of the $(n \cdot t)$ requested distinct memory cells on an n-processor DMM. The copies are distributed using c independently and randomly chosen functions from a $(2, \log^2 n)$-universal class of hash functions. We show that such a problem can be reduced in time $\mathcal{O}(t + \log^* n)$, w.h.p., to a constant number of independent "1 out of $c-1$" tasks, provided that $t \leq \sqrt[4]{\log n}$. Next we apply this reduction to obtain our time-processor optimal simulations of an $(n \cdot t)$-processor EREW PRAM on an n-processor DMM.

Let us consider the problem of accessing 1 out of the c copies of the $(n \cdot t)$ requested distinct memory cells, where the copies are distributed using independent hash functions $h_1, \ldots, h_c$. We assume that each processor of the DMM has a list of t keys (memory requests). We consider only the situation where $t \leq \sqrt[4]{\log n}$. We first use all-but-linear routing to satisfy all but $\mathcal{O}(n)$ of the access requests, by accessing the copies of the keys stored in memory modules given by h_1. Theorem 8.1 ensures that this phase can be performed in time $\mathcal{O}(t + \log^* n)$, w.h.p. Next, the remaining $\mathcal{O}(n)$ access requests are redistributed evenly among the processors of the DMM such that each processor gets $\mathcal{O}(1)$ of them. For that we use approximate prefix sums with respect to the number of nondelivered keys of each processor and then accordingly redistribute the keys. By Lemma 3.3, approximate prefix sums require $\mathcal{O}(\log^* n)$ time, w.h.p. Given computed approximate prefix sums, the distribution phase can be done in time $\mathcal{O}(t)$. Let us observe that, since the hash functions $h_2, h_3, \ldots, h_c$ are independent of h_1, the remaining $\mathcal{O}(n)$ requests are independent of $h_2, h_3, \ldots, h_c$. Therefore, we can use them as the input for a constant number of "1 out of $c-1$" tasks on the basis of hash functions $h_2, h_3, \ldots, h_c$. Thus the above algorithm reduces the initial problem to a constant number of "1 out of $c-1$" tasks in time $\mathcal{O}(t+\log^* n)$, w.h.p.

By our discussion in section 2, a simulation of one step of an $(n \cdot t)$-processor EREW PRAM on an n-processor DMM can be reduced to solving independently $\binom{a}{b-1}$ times (for $b > a/2$) the problem of accessing 1 out of the $a-b+1$ copies of the $(n \cdot t)$ requested distinct memory cells. Therefore, the algorithm above implies the following theorem.

THEOREM 9.1. *If one can execute a "1 out of $a-b$" task ($b > a/2$) in time bounded by $t \leq \sqrt[4]{\log n}$, w.h.p., then one step of an $(n \cdot t)$-processor EREW PRAM can be simulated on an n-processor DMM with delay $\mathcal{O}(\binom{a}{b-1} \cdot (t + \log^* n))$, w.h.p.*

Hence we can combine Theorem 9.1 with the algorithm for the "1 out of 2" task described in section 6.4 to obtain the following theorem.

THEOREM 9.2. *One step of an $(n \log\log\log n \log^* n)$-processor EREW PRAM can be simulated on an n-processor DMM with delay $\mathcal{O}(\log\log\log n \log^* n)$, w.h.p.*

We finally note that Karp, Luby, and Meyer auf de Heide [19] gave a slightly weaker theorem than Theorem 9.1 for CRCW PRAM simulations. We may apply it to get the following.

THEOREM 9.3. *One step of an $(n \log\log\log n \log^* n)$-processor CRCW PRAM can be simulated on an n-processor DMM with delay $\mathcal{O}(\log\log\log n (\log^* n)^2)$, w.h.p.*

10. A lower bound for the topological game. In this section we pinpoint the limits of our approach based on the "1 out of 2" task. We show that all previously known solutions based on the "1 out of 2" task as well as our algorithms are special cases of a game on the access graph, called the *topological game*. Then we prove a lower bound for the topological game.

All simulations based on the "1 out of 2" task, or equivalently on the analysis of the access game, have the following common scheme. In each step i each node v of H either analyzes its $k_{v,i}$-neighborhood or, on the basis of the structure of its $k_{v,i}$-neighborhood, decides which edge to remove, where $k_{v,i}$ is some integer. We can describe this scheme in a slightly more general way. Let k be the maximum of $k_{v,i}$ taken over all nodes v of H and all steps i of the algorithm. In the first superstep each node finds and analyzes its k-neighborhood. Then, in the remaining rounds it decides which edge to remove on the basis of the current k-neighborhood. Here we assume that each node will get the information on all removed edges in its k-neighborhood for free. It is easy to see that this extension makes the algorithm at least as powerful as the original one.

Let H be the access graph as defined in section 4 with n nodes and n/c edges, for $c < \log\log n$. Further assume that H is composed using two independent random hash functions. We define the *topological game* on the access graph H as follows. k is a parameter for the game.

Topological Game.

(1) Each node of H finds and analyzes its k-neighborhood; we charge for this $\log k$ steps.
(2) Repeat the following round until all the edges are removed:
   (a) Each node of degree one removes all edges in its k-neighborhood.
   (b) Each other node removes one incident edge. The choice of this edge solely depends on the topology of the current topology of its k-neighborhood.

We stress here the following features of this scheme.

Explanation and discussion of step 10. Let us observe that in all previous simulations [8, 12, 19, 22, 25] only a 1-neighborhood was analyzed. On the other hand, we notice that the time needed for computing any information about the k-neighborhood of a node v that involves the knowledge on any node that is in distance k from v seems to require $\Omega(\log k)$ steps. We remark that our data structures presented in section 6 allow each processor to obtain useful information about its k-neighborhood in $\mathcal{O}(\log k \cdot \log^* n)$ time, w.h.p.

Explanation and discussion of step 0a. This step is motivated by the fact that using a slight modification of the algorithm Cleaning Up Connected Components, each node that is close to the border of its connected component (that is, a node whose k-neighborhood contains a simple path of length less than k that cannot be extended by any edge to a simple path) can be removed in constant time using only the information about the k-neighborhood.

Explanation and discussion of step 0b. Here we make a key assumption concerning the topological game. A node bases its decision of which edge to choose on the topologies of the $(k-1)$-neighborhoods of its (direct) neighbors. The names of the nodes and the labels of the edges are not allowed to be used. In particular, in the case of all $(k-1)$-neighborhoods being disjoint and isomorphic, the node can only choose a random incident edge to remove. We further assume that removing further edges during the game does not increase the number of rounds required. All strategies known so far fit into this model. Especially, no simulation strategy is known that takes advantage of using the labels of nodes and edges to break ties, i.e., to choose among neighbors with disjoint isomorphic $(k-1)$-neighborhoods. On the other hand, it is challenging to find simulations that take advantage of using the labels, or to extend our lower bound.

The main idea of the lower bound for the topological game is to focus only on fully

symmetric structures in H. We show that there exists a node that has a (topologically) symmetric k-neighborhood, possibly after removing some edges, and does not contain nodes of degree one in its k-neighborhood. Then we assume that it randomly makes the decision which outgoing edge to remove. We show that, after performing these random decisions, a smaller symmetric subgraph will be left in the next round, with sufficiently high probability. The bound for decreasing the size of the symmetric subgraph will yield the lower bound.

For $0 \le i < \frac{\log\log\log n}{8\log\log\log\log n}$, define the values of δ_i and d_i as follows:

$$\delta_i = \frac{\log\log n}{(\log\log\log n)^{4(i+1)}} \quad \text{and} \quad d_i = \frac{\log\log n}{(\log\log\log n)^{4(i+1)-2}} \ .$$

We will use the following inequalities for values δ_i and d_i.

LEMMA 10.1. *For every* $1 \le k \le \sqrt{\log\log n}$, $0 \le i < \frac{\log\log\log n}{8\log\log\log\log n} - 1$, *and sufficiently large* n *the following inequalities hold:*

(i) $2e^{-\frac{2\cdot\delta_i^{d_i}}{2^{2(\delta_i-\delta_{i+1})}}} \le \frac{1}{\log n}$ *and*

(ii) $\frac{3\cdot\delta_i^{d_i}}{2^{\delta_i-\delta_{i+1}-1}} \le \delta_i^{d_i-d_{i+1}-2k}$.

Proof.

(i) It is enough to show that

$$d_i \log\delta_i - 2\delta_i + 2\delta_{i+1} \ge \log\ln(2\log n).$$

If we substitute the terms that define δ_i and d_i on the left-hand side and use the assumption about the range of i, we get the following inequality:

$$\begin{aligned}
&\frac{\log\log n}{(\log\log\log n)^{4(i+1)-2}}(\log\log\log n - 4(i+1)\log\log\log\log n)\\
&-2\left(\frac{\log\log n}{(\log\log\log n)^{4(i+1)}} + \frac{\log\log n}{(\log\log\log n)^{4(i+2)}}\right)\\
&\ge \frac{\log\log n}{(\log\log\log n)^{4(i+1)-2}}\\
&\ge \sqrt{\log\log n}\,(\log\log\log n)^2\\
&\ge \log\ln(2\log n).
\end{aligned}$$

(ii) It is enough to show that

$$\delta_i - \delta_{i+1} - 3 - d_{i+1}\log\delta_i - 2k\log\delta_i \ge 0.$$

It is easily checked that $2k \le d_{i+1}$ for all i in the indicated range. This yields the following inequalities if we substitute the terms that define δ_i and d_i on the left-hand side and use the assumption about the range of i and k:

$$\begin{aligned}
&\delta_i - \delta_{i+1} - 3 - d_{i+1}\log\delta_i - 2k\log\delta_i\\
&\ge \delta_i - \delta_{i+1} - 3 - 2d_{i+1}\log\delta_i\\
&= \frac{\log\log n}{(\log\log\log n)^{4(i+1)}} - \frac{\log\log n}{(\log\log\log n)^{4(i+2)}} - 3\\
&\quad -2\frac{\log\log n}{(\log\log\log n)^{4(i+2)-2}}\log\left(\frac{\log\log n}{(\log\log\log n)^{4(i+1)}}\right)
\end{aligned}$$

$$\geq \frac{\log\log n}{2(\log\log\log n)^{4(i+1)}} - 3 - 2\frac{\log\log n}{(\log\log\log n)^{4(i+2)-3}}$$
$$\geq \frac{\log\log n}{4(\log\log\log n)^{4(i+1)}} \geq 0 \ . \qquad \square$$

DEFINITION 8. *A (δ_i, d_i)-tree is an undirected, unlabeled, and acyclic connected graph that contains a vertex r of degree δ_i such that all vertices in distance (exactly) d_i from r are of degree 1 and all other vertices are of degree δ_i.*

Fix an algorithm A that performs the topological game. We say a labeled directed graph A contains a copy of an unlabeled undirected graph B if, after removal of the labels and the edge orientation in A, the obtained graph contains a subgraph isomorphic to B. We want to maintain the condition that the access graph remaining after performing i rounds of the algorithm contains a copy of a (δ_i, d_i)-tree with sufficiently high probability. The proof of this invariant is done by induction, which is based on the following two lemmas.

LEMMA 10.2. *Let G be the directed access graph with n labeled nodes and n/c labeled edges (cf. section 4.1). Let $c < \log\log n$, and let $\mathfrak{T}_q$ be a fixed unlabeled and undirected tree with q nodes. For $q \leq \frac{\log n}{10\log\log n}$ and sufficiently large n, the probability that G contains a copy of $\mathfrak{T}_q$ is at least $1/2$.*

The proof of the lemma is in the spirit of the proof of a similar lemma for balanced graphs due to Erdős and Rényi [9].

Proof. Since the lemma trivially holds for $q = 1$, we assume that $q \geq 2$. Let $\mathcal{T}_q^{(n)}$ be the set of all subgraphs of the complete directed graph on n nodes and n/c edges labeled by the elements of S (cf. section 4.1) that, after removal of the labels and the edge orientations, are isomorphic to $\mathfrak{T}_q$. With each $T \in \mathcal{T}_q^{(n)}$ we associate a binary random variable $\mathcal{E}(T)$, such that $\mathcal{E}(T) = 1$ or $\mathcal{E}(T) = 0$ according to whether T is a subgraph of G or not. Our aim is to bound the probability that $\sum_{T\in\mathcal{T}_q^{(n)}} \mathcal{E}(T) = 0$. We estimate that probability by using Chebyshev's inequality. For that purpose we now provide bounds for the expectation and the variance of the random variable $\sum_{T\in\mathcal{T}_q^{(n)}} \mathcal{E}(T)$.

First consider the expected value $E(\sum \mathcal{E}(T))$. The probability that a fixed tree with $q-1$ edges is a subgraph of G is $(1/n^2)^{q-1}$. In order to obtain a lower bound on the number of trees in $\mathcal{T}_q^{(n)}$ we observe that we may obtain a subset of $\mathcal{T}_q^{(n)}$ by selecting trees with given q nodes and given labels assigned to the edges of the tree. Hence we obtain the following formula:

$$E\left(\sum_{T\in\mathcal{T}_q^{(n)}} \mathcal{E}(T)\right) = \sum_{T\in\mathcal{T}_q^{(n)}} E(\mathcal{E}(T))$$
$$= |\mathcal{T}_q^{(n)}| \cdot \left(\frac{1}{n^2}\right)^{q-1} \tag{1}$$
$$\geq \binom{n}{q} \frac{\left(\frac{n}{c}\right)!}{\left(\frac{n}{c} - (q-1)\right)!} \left(\frac{1}{n^2}\right)^{q-1}$$
$$\geq \left(\frac{n}{q}\right)^q \left(\frac{n}{2c}\right)^{q-1} n^{-2(q-1)}$$
$$\geq \frac{n}{(2cq)^q} \ . \tag{2}$$

Let T_1 and T_2 be two elements from $\mathcal{T}_q^{(n)}$. If T_1 and T_2 have no edges in common, then

$$E(\mathcal{E}(T_1)\mathcal{E}(T_2)) = \left(\frac{1}{n^2}\right)^{2(q-1)} .$$

Therefore

$$E\left(\sum_{\substack{T_1,T_2\in\mathcal{T}_q^{(n)}\\ T_1,T_2\\ \text{edge-disjoint}}} \mathcal{E}(T_1)\mathcal{E}(T_2)\right) = |\{T_1,T_2\in\mathcal{T}_q^{(n)} : T_1,T_2 \text{ edge disjoint}\}| \cdot \left(\frac{1}{n^2}\right)^{2(q-1)}$$

$$\leq (|\mathcal{T}_q^{(n)}|)^2 \cdot \left(\frac{1}{n^2}\right)^{2(q-1)}$$

$$\overset{(1)}{\leq} \left(E\left(\sum_{T\in\mathcal{T}_q^{(n)}} \mathcal{E}(T)\right)\right)^2 . \tag{3}$$

If T_1 and T_2 have exactly r edges in common ($1 \leq r \leq q-1$), then we get

$$E(\mathcal{E}(T_1)\mathcal{E}(T_2)) = \left(\frac{1}{n^2}\right)^{2(q-1)-r} .$$

Because T_1 and T_2 are trees, they have at least $r+1$ nodes in common. Hence, we can give an upper bound for the number of such pairs T_1, T_2 of subgraphs:

$$\begin{aligned}
&(2^q \cdot q^{q-2})^2 \sum_{j=r+1}^{q} \binom{n}{q}\binom{q}{j}\binom{n-q}{q-j}\left(\frac{n}{c}\right)^{2(q-1)-r} \\
&\leq q^{4q-4} \sum_{j=r+1}^{q} \frac{n!}{q!(n-q)!}\frac{q!}{j!(q-j)!}\frac{(n-q)!}{(q-j)!(n-2q+j)!}\left(\frac{n}{c}\right)^{2(q-1)-r} \\
&\leq q^{4q-4} \sum_{j=r+1}^{q} \frac{n^{2q-j+2(q-1)-r}}{j!((q-j)!)^2 c^{2(q-1)-r}} \\
&\leq q^{4q-4} q n^{4q-2r-3} .
\end{aligned}$$

Hence we obtain

$$E\left(\sum_{\substack{T_1,T_2\in\mathcal{T}_q^{(n)}\\ T_1,T_2\\ \text{not edge-disjoint}}} \mathcal{E}(T_1)\mathcal{E}(T_2)\right) \leq \sum_{r=1}^{q-1} q^{4q-3} n^{4q-2r-3}\left(\frac{1}{n^2}\right)^{2(q-1)-r} \leq nq^{4q-2} . \tag{4}$$

Now we may bound $E((\sum_{T\in\mathcal{T}_q^{(n)}} \mathcal{E}(T))^2)$:

$$E\left(\left(\sum_{T\in\mathcal{T}_q^{(n)}} \mathcal{E}(T)\right)^2\right)$$

$$
\begin{aligned}
&= E\left(\sum_{T\in\mathcal{T}_q^{(n)}}\mathcal{E}(T)\right) + E\left(\sum_{\substack{T_1,T_2\in\mathcal{T}_q^{(n)}\\ T_1,T_2\\ \text{edge-disjoint}}}\mathcal{E}(T_1)\mathcal{E}(T_2)\right)\\
&\quad + E\left(\sum_{\substack{T_1,T_2\in\mathcal{T}_q^{(n)}\\ T_1,T_2\\ \text{not edge-disjoint}}}\mathcal{E}(T_1)\mathcal{E}(T_2)\right)\\
&\overset{(3)(4)}{\leq} E\left(\sum_{T\in\mathcal{T}_q^{(n)}}\mathcal{E}(T)\right) + \left(E\left(\sum_{T\in\mathcal{T}_q^{(n)}}\mathcal{E}(T)\right)\right)^2 + nq^{4q-2}\ . \qquad (5)
\end{aligned}
$$

Finally we provide an upper bound for the variance:

$$
\begin{aligned}
\operatorname{Var}\left(\sum_{T\in\mathcal{T}_q^{(n)}}\mathcal{E}(T)\right) &= E\left(\left(\sum_{T\in\mathcal{T}_q^{(n)}}\mathcal{E}(T)\right)^2\right) - \left(E\left(\sum_{T\in\mathcal{T}_q^{(n)}}\mathcal{E}(T)\right)\right)^2\\
&\overset{(5)}{\leq} E\left(\sum_{T\in\mathcal{T}_q^{(n)}}\mathcal{E}(T)\right) + nq^{4q-2}\ . \qquad (6)
\end{aligned}
$$

Using the Chebyshev inequality we can bound the probability that no $T \in \mathcal{T}_q^{(n)}$ is a subgraph of G:

$$
\begin{aligned}
\Pr\left(\sum_{T\in\mathcal{T}_q^{(n)}}\mathcal{E}(T)=0\right) &\leq \frac{\operatorname{Var}\left(\sum\mathcal{E}(T)\right)}{\left(E\left(\sum\mathcal{E}(T)\right)\right)^2}\\
&\overset{(6)}{\leq} \frac{\sum E(\mathcal{E}(T)) + nq^{4q-2}}{\left(E\left(\sum\mathcal{E}(T)\right)\right)^2}\\
&\overset{(2)}{\leq} \frac{(2cq)^q}{n} + \frac{nq^{4q-2}(2cq)^{2q}}{n^2}\\
&= \frac{(2cq)^q + (2c)^{2q}q^{6q-2}}{n}\\
&\leq \frac{q^{10q}}{n} \qquad \text{for} \quad c \leq q\\
&\leq \frac{1}{2} \qquad \text{for} \quad q \leq \frac{\log n}{10\log\log n}\ .
\end{aligned}
$$

Therefore, the probability that G contains a copy of $\mathfrak{T}_q$ is at least $1/2$ for $q \leq \frac{\log n}{10\log\log n}$. □

LEMMA 10.3. *Consider a (δ_i, d_i)-tree for $0 \leq i < \frac{\log\log\log n}{8\log\log\log\log n}$, and let $k \leq \sqrt{\log\log n}$. If every node randomly removes an incident edge, then, with probability at least $1 - 1/\log n$, at most $\delta_i^{d_i - d_{i+1} - 2k}$ of the nodes have degree smaller than δ_{i+1}.*

Proof. Consider a fixed node v with degree δ_i. Each edge incident to v will be removed by an incident node with the same probability $1/\delta_i$. Define for $1 \leq j \leq \delta_i$ the binary random variable X_j which is one if and only if the jth incident edge of v will be removed by an incident node. We know that $\Pr(X_j = 1) = 1/\delta_i$. Let $X = \sum X_j$. Note that $E(X) = 1$. We want to bound the deviation from the expected value. Because the decisions of the adjacent nodes are independent, we can use the Chernoff bound [15]:

$$\Pr(X \geq \alpha) = \Pr(X \geq \alpha E(X)) \leq \frac{1}{2^\alpha} \text{ for } \alpha \geq 5 \ .$$

In particular, this implies that

$$\Pr(\text{a node has degree smaller than } \delta_{i+1}) \leq \left(\frac{1}{2}\right)^{\delta_i - \delta_{i+1} - 1} \ .$$

Let x_l, $1 \leq l \leq \frac{\delta_i^{d_i+1}-1}{d_i-1}$, be the sequence of independent random decisions made by the $\frac{\delta_i^{d_i+1}-1}{d_i-1}$ nodes of the (δ_i, d_i)-tree. The random variable

$$Y = f\left(\left\{x_l : 1 \leq l \leq \frac{\delta_i^{d_i+1}-1}{d_i-1}\right\}\right)$$

denotes the number of nodes that have degree smaller than δ_{i+1}. If we change the value of one x_l, Y can be changed by at most two. The expected number of nodes having degree less than δ_{i+1} can be bounded by

$$\frac{\frac{\delta_i^{d_i+1}-1}{d_i-1}}{2^{\delta_i-\delta_{i+1}-1}} \leq 2\frac{\delta_i^{d_i}}{2^{\delta_i-\delta_{i+1}-1}} \ .$$

Using the method of bounded differences (Lemma 3.1), we get

$$\begin{aligned}
\Pr\left(Y \geq 3\frac{\delta_i^{d_i}}{2^{\delta_i-\delta_{i+1}-1}}\right) &\leq \Pr\left(Y \geq E(Y) + \frac{\delta_i^{d_i}}{2^{\delta_i-\delta_{i+1}-1}}\right) \\
&\leq 2e^{-2\frac{\left(\delta_i^{d_i}\right)^2}{2^{2(\delta_i-\delta_{i+1}-1)}} \cdot \frac{d_i-1}{4\left(\delta_i^{d_i+1}-1\right)}} \\
&\leq 2e^{-\frac{\delta_i^{d_i}}{2^{2(\delta_i-\delta_{i+1})}}} \ .
\end{aligned}$$

Substituting the values of δ_i and d_i into this formula and using Lemma 10.1 we get that for $k \leq \sqrt{\log\log n}$ the number of nodes with small degree can be bounded by $\delta_i^{d_i-d_{i+1}-2k}$ with probability at least $1 - 1/\log n$. □

Using these two lemmas we can show the following.

LEMMA 10.4. *After performing i rounds of the algorithm A, a (δ_i, d_i)-tree is a subgraph of the remaining access graph with probability at least $(1 - 1/\log n)^i$ for $1 \leq i < \frac{\log\log\log n}{8\log\log\log\log n}$ and $k \leq \sqrt{\log\log n}$.*

Proof. The proof is done by induction on i. For $i = 0$ we use Lemma 10.2. Assume that the lemma holds for some i, where $i < \frac{\log\log\log n}{8\log\log\log\log n}$. From the induction hypothesis we know that a (δ_i, d_i)-tree is a subgraph of the access graph at the

beginning of round $i+1$. We consider only the edges from this tree and remove all other edges. Because of the definition of the topological game, we remove all nodes that are within distance at most k from a leaf of the tree. Each decision based on the topology of the k-neighborhood made by the remaining nodes is random and is independent of decisions of other nodes.

Hence, using Lemma 10.3, at most $\delta_i^{d_i-d_{i+1}-2k}$ nodes have degree smaller than δ_{i+1}, and a (δ_i, d_i)-tree is a subgraph of the remaining graph. □

The invariant of Lemma 10.4 holds for k small enough in every round i, $1 \le i < \frac{\log\log\log n}{8\log\log\log\log n}$, even if we start only with $\frac{n}{\log\log\log n}$ edges. This implies the following theorem.

THEOREM 10.5. *The expected number of rounds in any topological game until all edges of the access graph H will be removed is $\Omega(\frac{\log\log\log n}{\log\log\log\log n})$.*

Proof. After $\frac{\log\log\log n}{8\log\log\log\log n}$ rounds it is only possible to pick a k-neighborhood for

$$k \le 2^{\frac{\log\log\log n}{8\log\log\log\log n}} \le \sqrt{\log\log n}\ .$$

Therefore, when we use Lemma 10.4, after $i \le \frac{\log\log\log n}{8\log\log\log\log n}$ rounds some edges will be left. The probability for this event can be bounded by

$$\left(1-\frac{1}{\log n}\right)^{\frac{\log\log\log n}{8\log\log\log\log n}} \ge 1/e\ . \qquad \square$$

REFERENCES

[1] R. J. ANDERSON AND G. L. MILLER, *Optical Communication for Pointer Based Algorithms*, Tech. Report CRI 88-14, University of Southern California, Los Angeles, CA, 1988.

[2] H. BAST AND T. HAGERUP, *Fast and reliable parallel hashing*, in Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures, ACM Press, New York, NY, 1991, pp. 50–61.

[3] H. BAST AND T. HAGERUP, *Fast parallel space allocation, estimation and integer sorting*, Inform. and Comput., 123 (1995), pp. 72–110.

[4] O. BERKMAN AND U. VISHKIN, *Recursive star-tree parallel data structure*, SIAM J. Comput., 22 (1993), pp. 221–242.

[5] J. L. CARTER AND M. N. WEGMAN, *Universal classes of hash functions*, J. Comput. System Sci., 18 (1979), pp. 143–154.

[6] M. DIETZFELBINGER, A. KARLIN, K. MEHLHORN, F. MEYER AUF DER HEIDE, H. ROHNERT, AND R. E. TARJAN, *Dynamic perfect hashing: Upper and lower bounds*, SIAM J. Comput., 23 (1994), pp. 738–761.

[7] M. DIETZFELBINGER AND F. MEYER AUF DER HEIDE, *How to distribute a dictionary in a complete network*, in Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, ACM Press, New York, NY, 1990, pp. 117–127.

[8] M. DIETZFELBINGER AND F. MEYER AUF DER HEIDE, *Simple, efficient shared memory simulations*, in Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures, ACM Press, New York, NY, 1993, pp. 110–119.

[9] P. ERDŐS AND A. RÉNYI, *On the evolution of random graphs*, Magyar Tud. Akad. Mat. Kut. Int. Közl., 5 (1960), pp. 17–61.

[10] J. GIL, Y. MATIAS, AND U. VISHKIN, *Towards a theory of nearly constant time parallel algorithms*, in Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science, IEEE Press, Los Alamitos, CA, 1991, pp. 698–710.

[11] L. A. GOLDBERG, M. JERRUM, T. LEIGHTON, AND S. RAO, *Doubly logarithmic communication algorithms for optical communication parallel computers*, SIAM J. Comput., 26 (1997), pp. 1100–1119.

[12] L. A. GOLDBERG, Y. MATIAS, AND S. RAO, *An optical simulation of shared memory*, in Proceedings of the 6th Annual ACM Symposium on Parallel Algorithms and Architectures, ACM Press, New York, NY, 1994, pp. 257–267.

[13] M. T. Goodrich, Y. Matias, and U. Vishkin, *Optimal parallel approximation algorithms for prefix sums and integer sorting*, in Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms, ACM Press, New York, NY, 1994, pp. 241–250.

[14] T. Hagerup, *The log-star revolution*, in Proceedings of the 9th Annual Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Comput. Sci. 577, Springer-Verlag, Heidelberg, 1992, pp. 259–278.

[15] T. Hagerup and C. Rüb, *A guided tour of Chernoff bounds*, Inform. Process. Lett., 33 (1989/90), pp. 305–308.

[16] J. JáJá, *An Introduction to Parallel Algorithms*, Addison Wesley, Reading, MA, 1992.

[17] A. Karlin and E. Upfal, *Parallel hashing—an efficient implementation of shared memory*, in Proceedings of the 18th Annual ACM Symposium on Theory of Computing, ACM Press, New York, NY, 1986, pp. 160–168.

[18] R. Karp and V. Ramachandran, *A survey of parallel algorithms for shared-memory machines*, in Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity, J. van Leeuwen, ed., Elsevier Science, Amsterdam, The Netherlands, 1990, pp. 869–941.

[19] R. M. Karp, M. Luby, and F. Meyer auf der Heide, *Efficient PRAM simulation on a distributed memory machine*, Algorithmica, 16 (1996), pp. 517–542.

[20] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[21] T. Leighton, *Methods for message routing in parallel machines*, in Proceedings of the 24th Annual ACM Symposium on Theory of Computing, ACM Press, New York, NY, 1992, pp. 77–96.

[22] P. D. MacKenzie, C. G. Plaxton, and R. Rajaraman, *On contention resolution protocols and associated probabilistic phenomena*, J. ACM, 45 (1998), pp. 324–378.

[23] C. McDiarmid, *On the method of bounded differences*, in Surveys in Combinatorics, J. Siemons, ed., London Math. Soc. Lecture Note Ser. 141, Cambridge University Press, Cambridge, UK, 1989, pp. 148–188.

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

[25] F. Meyer auf der Heide, C. Scheideler, and V. Stemann, *Exploiting storage redundancy to speed up randomized shared memory simulations*, Theoret. Comput. Sci., 162 (1996), pp. 245–281.

[26] P. Ragde, *The parallel simplicity of compaction and chaining*, J. Algorithms, 14 (1993), pp. 371–380.

[27] A. G. Ranade, *How to emulate shared memory*, J. Comput. System Sci., 42 (1991), pp. 307–326.

[28] J. H. Reif, ed., *Synthesis of Parallel Algorithms*, Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[29] A. Siegel, *On universal classes of fast high performance hash functions, their time-space tradeoff, and their applications*, in Proceedings of the 30th IEEE Symposium on Foundations of Computer Science, IEEE Press, Los Alamitos, CA, 1989, pp. 20–25.

[30] V. Stemann, *Contention Resolution in Hashing Based Shared Memory Simulations*, Ph.D. thesis, Fachbereich 17, University of Paderborn, Paderborn, Germany, 1995.

[31] E. Upfal, *Efficient schemes for parallel communication*, J. ACM, 31 (1984), pp. 507–517.

[32] E. Upfal and A. Wigderson, *How to share memory in a distributed system*, J. ACM, 34 (1987), pp. 116–127.

[33] L. G. Valiant, *General purpose parallel architectures*, in Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity, J. van Leeuwen, ed., Elsevier Science, Amsterdam, The Netherlands, 1990, pp. 943–971.