

Shared Memory Simulations with Triple-Logarithmic Delay ^{*}

(Extended Abstract)

Artur Czumaj¹, Friedhelm Meyer auf der Heide², and Volker Stemann¹

¹ Heinz Nixdorf Institute, University of Paderborn,
D-33095 Paderborn, Germany

² Heinz Nixdorf Institute and Department of Computer Science,
University of Paderborn, D-33095 Paderborn, Germany

Abstract. We consider the problem of simulating a PRAM on a distributed memory machine (DMM). Our main result is a randomized algorithm that simulates each step of an n -processor CRCW PRAM on an n -processor DMM with $\mathcal{O}(\log \log \log n \log^* n)$ delay, with high probability. This is an exponential improvement on all previously known simulations. It can be extended to a simulation of an $(n \log \log \log n \log^* n)$ -processor EREW PRAM on an n -processor DMM with optimal delay $\mathcal{O}(\log \log \log n \log^* n)$, with high probability. Finally a lower bound of $\Omega(\log \log \log n / \log \log \log \log n)$ expected time is proved for a large class of randomized simulations that includes all known simulations.

1 Introduction

Parallel machines that communicate via a shared memory (*Parallel Random Access Machines, PRAMs*) are the most commonly used machine model for describing parallel algorithms (see e.g. [J92]). The PRAM is relatively comfortable to program, because the programmer does not have to deal with the hardware limitations, like e.g. synchronization, data locality or interprocessor communication, and can only focus on the combinatorial properties of the problem at hand. On the other hand shared memory machines are very unrealistic from the technological point of view, because, for example, on large machines a parallel shared memory access can only be realized at the cost of a significant time delay. A more realistic model which tries to overcome this unrealistic assumption, is the *Distributed Memory Machine (DMM)*, in which the memory is divided into a limited number of memory modules, one module per processor. Each such module can respond to only one access 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.

^{*} Supported in part by DFG-Graduiertenkolleg “Parallele Rechnernetze in der Produktionstechnik”, ME 872/4-1, by DFG-Sonderforschungsbereich 1511 “Massive Parallelität: Algorithmen, Entwurfsmethoden, Anwendungen”, by DFG Leibniz Grant Me872/6-1, and by the Esprit Basic Research Action Nr 7141 (ALCOM II)

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, and packet routing is used to access the modules [R91, L92a, L92b, U84, KU86]. In this paper we study DMMs with a complete interconnection between processors and modules.

Simulations based on hashing distribute the shared memory cells U among the modules using one or more hash functions $h_i : U \rightarrow [n]$,³ $i \in [a]$; cell $u \in U$ is stored in the modules $M_{h_1(u)}, \dots, M_{h_a(u)}$. All such simulations assume that h_1, \dots, h_a are randomly chosen from a *high performance universal class* of hash functions as e.g. presented by Dietzfelbinger and Meyer auf der Heide [DM90] or Siegel [S89]. The *delay* of a simulation is the time needed to simulate a parallel memory access of a PRAM. We say a randomized simulation of a p -processor PRAM on an n -processor DMM is *time-processor optimal* if the delay is $\mathcal{O}(p/n)$ with high probability (w.h.p.)⁴. It is easily seen that a simulation of an n -processor EREW PRAM on an n -processor DMM using one hash function has contention $\Omega(\log n / \log \log n)$, w.h.p., even if the hash function behaves like a random function. Karp et al. [KLM92] invent the idea of performing a simulation based on more than one hash function. They present a simulation of an n -processor EREW PRAM on an n -processor DMM with delay $\mathcal{O}(\log \log n)$, w.h.p., with three hash functions. Thus, at the expense of increasing the total storage requirement by a constant factor, the running time of the simulation is exponentially decreased. They also obtain a time-processor optimal simulation of an $(n \log \log n \log^* n)$ -processor EREW PRAM on an n -processor DMM. Using the majority technique due to Upfal and Wigderson [UW87], Dietzfelbinger and Meyer auf der Heide [DM93] extend this result to a much simpler schedule for an $\mathcal{O}(\log \log n)$ simulation on the weaker c -collision DMM. In this model a module can only answer, if it gets less than c requests; otherwise it sends a collision symbol. Goldberg et al. [GMR94] show that one can perform a time-processor optimal simulation with delay $\mathcal{O}(\log \log n)$, w.h.p., even on a 1-collision model (called also OCPC). Meyer auf der Heide et al. [MSS95] extend the simulation on a DMM to more hash functions which yields a delay of $\mathcal{O}(\log \log n / \log \log \log n)$, w.h.p.. As it uses a non-constant number of hash functions it cannot be turned into a time-processor optimal simulation.

The techniques used in these papers seem not to yield simulations with smaller delay. In particular, MacKenzie et al. [MPR94] and independently Meyer auf der Heide et al. [MSS95] show lower bounds for classes of algorithms that capture all these algorithms. To break the $\Omega(\log \log n)$ lower bound of Meyer auf der Heide et al. [MSS95] for simulations with constant memory redundancy one has to use non-oblivious techniques. Using information about the number of accesses at a module, Czumaj et al. [CMS95] present a randomized simulation of an EREW PRAM with delay $\mathcal{O}(\log \log n / \log \log \log n)$ and constant memory redundancy, w.h.p.. They also extend this result to a time-processor optimal

³ In this paper $[n]$ will always denote the set $\{1, 2, \dots, n\}$.

⁴ W.h.p. means “with probability at least $1 - n^{-l}$ for any constant l ”.

simulation of an $(n \log \log n \log^* n / \log \log \log n)$ -processor EREW PRAM on an n -processor DMM.

In this paper we design new shared memory simulations that improve all previously known results by an exponential decrease of the delay. We design a simulation of an n -processor EREW PRAM on an n -processor DMM with delay $O(\log \log \log n \log^* n)$, w.h.p.. We model access requests by an access graph, in which the nodes correspond to the modules and the edges correspond to the accesses of the processors. In all previously known simulations each module essentially decides which request to answer based only on the structure of the corresponding node, its incident edges, and its neighbors. We show that using as much information about the structure of the neighborhood of each node as available during the time of the simulation may drastically improve simulations. A clever access protocol ensures that our access graph will decay in small independent subgraphs. Within these small components we can perform a constant time protocol to answer all remaining accesses of the processors. The algorithms are based on sophisticated log-star techniques that explore the neighborhood of each node and an analysis of the random structure of the access graph. In a full paper we present a simple protocol which enables us to turn any simulation with constant storage overhead of an n -processor EREW PRAM on an n -processor DMM into a time-processor optimal one. Finally we show that our implementations are almost optimal. We present an $\Omega(\log \log \log n / \log \log \log \log n)$ lower bound for a large class of randomized simulations that includes all the known simulations.

Note that our simulations of an n -processor EREW PRAM on an n -processor DMM can be easily extended to simulations of an n -processor CRCW PRAM without increasing the delay. We are not able, however, to obtain a time-processor optimal simulation of a CRCW PRAM. Although the techniques of Karp et al. [KLM92] can be used to get a simulation of an $\mathcal{O}(n \log \log \log n \log^* n)$ -processor CRCW PRAM on an n -processor DMM with delay $\mathcal{O}(\log \log \log n (\log^* n)^2)$, which is only a factor of $\log^* n$ away from optimality.

The paper is organized as follows. In Section 2 we proceed with the definition of the computation models and state a graph-theoretic lemma that is the basis of all our analysis. In Section 3 we elaborate some algorithmic utilities for a random graph, especially how to explore the k -neighborhood of a node, which is essential for our algorithms. In Section 4 we present the $\mathcal{O}(\log \log \log n \log^* n)$ simulation. Finally Section 5 presents the $\Omega(\log \log \log n / \log \log \log \log n)$ lower bound for simulations on the DMM.

Because of space limitations some details and proofs are omitted in this extended abstract.

2 Preliminaries

A *parallel random access machine (PRAM)* consists of p processors P_1, \dots, P_p and a shared memory with cells $U = [m]$. The processors work synchronously and have random access to the shared memory cells, each of which can store

an integer. We consider two models of a PRAM, an *exclusive read exclusive write (EREW)* PRAM, in which concurrent reads and writes are forbidden, and a *concurrent read concurrent write (CRCW)* PRAM, which allows concurrent reads and writes. We only deal with an ARBITRARY CRCW PRAM, in which if several processors want to write to the same memory cell simultaneously an arbitrary one succeeds.

A *distributed memory machine (DMM)* has n processors, Q_1, \dots, Q_n , which communicate via a distributed memory consisting of n modules, M_1, \dots, M_n . Each module has a communication window. A module can read from or write into its window. From the point of view of the processors, a window acts like a shared memory cell, where concurrent accesses are allowed. If more than one processor want to access the same module simultaneously then an arbitrary one succeeds. The following lemma can easily be obtained.

Lemma 1. *An n -processor ARBITRARY CRCW PRAM with $\mathcal{O}(n)$ shared memory cells and an n -processor DMM can simulate each other with constant delay.*

Our PRAM simulations follow the ideas of Upfal and Wigderson [UW87], and Dietzfelbinger and Meyer auf der Heide [DM93] (see also [M92]). The memory of a PRAM is hashed using three hash functions h_1, h_2 , and h_3 . That means, each memory cell $u \in U$ of a PRAM will be stored in the modules $M_{h_1(u)}, M_{h_2(u)}$, and $M_{h_3(u)}$ of a DMM. We will call the representations of u in the $M_{h_i(u)}$'s the copies of u . For simplicity of presentation we assume that all the hash functions used are random functions. Universal classes of functions developed by Siegel [S89] are sufficient for our purposes, however (see e.g. [KLM92, MSS95, GMR94]). For the simulation of a PRAM step we use the technique of Upfal and Wigderson [UW87] which ensures that it suffices to access arbitrary two out of the three copies of a shared memory to guarantee a correct simulation. To write to a memory cell a processor of the DMM accesses at least two of the copies and adds a time stamp to them indicating the (PRAM-) time of the update. To read a memory cell a processor has to access two of the copies and takes the one with the latest time stamp.

As in [CMS95], we modify the two-out-of-three idea and split this schedule into three steps of trying to access one out of two copies with a different pair of hash functions in each step. Clearly in this way we always access at least two copies. Therefore in the following we will analyze how to simulate an access of the shared memory that uses 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/c , for some constant $c \geq 1$, which will be specified in Lemma 2. 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 us call a schedule where one has to access for each $u \in S$ at least one of the two possible copies a *one-out-of-two schedule*. Let $G = ([n], E)$ be the labeled directed graph, defined by h_1, h_2 and the set of requests S , that 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 . Let H be the labeled graph obtained from G by removing all directions from the edges.

Our simulations rely on the properties of the random graph H . The following lemma was proved partially by Karp et al. [KLM92] and Czumaj et al. [CMS95].

Lemma 2. *For each constant l there is a constant c such that for the graph H , consisting of n nodes and n/c edges, the following conditions hold with probability at least $1 - 1/n^l$.*

- (a) H has no connected component of size larger than $\log n$.
- (b) For each pair of nodes v and w there are only $O(1)$ simple paths from v to w .
- (c) There are only $O(1)$ cycles in H .
- (d) For each $\xi < \log c/4l$

$$\sum_{\text{connected components } C} |C| \cdot 2^{|C| \xi} = O(n) .$$

3 Algorithmic Utilities

All the algorithms we describe in this section are designed on an ARBITRARY CRCW PRAM with $\mathcal{O}(n)$ space. Lemma 1 shows that they run on a DMM as well, with constant delay.

3.1 Log-Star Techniques

The following are the main tools used by our algorithms. Let $b > 1$ be any small constant, e.g. $b = 2$. If an array of size bn contains at least n objects we will call it *padded-consecutive*.

Given n integers x_1, x_2, \dots, x_n , the *strong semisorting* problem [BH93] is to store them in a padded-consecutive array, such that all variables with the same value occur in a padded-consecutive subarray. Given n bits x_1, x_2, \dots, x_n , the *chaining* problem [R93, BV93] is to find for each x_i , the nearest 1's both to its left and to its right. The *processor allocation* problem [GMV91] is to redistribute m tasks among n processors, so that each processor gets $O(1 + m/n)$ tasks. For a given sequence of integers, x_1, \dots, x_n , $x_i \in [n]$, the *approximate parallel prefix sums* problem [GMV94] is to find a sequence $y_0 = 0, y_1, \dots, y_n$, $y_i \in [n]$, such that for $i \in [n]$, $x_i \leq y_i - y_{i-1} \leq bx_i$. We combine results of [R93, BV93, GMV91, GMV94, BH93] in the following lemma.

Lemma 3. *The strong semisorting problem, the chaining problem, the processor allocation problem, and the approximate parallel prefix sums can be solved on an ARBITRARY CRCW PRAM in $\mathcal{O}(\log^* n)$ time with linear total work and linear space, with probability at least $1 - 2^{-n^\epsilon}$ for some constant $\epsilon > 0$.*

3.2 Algorithms on the Random Graph H

Throughout this section H will always denote a graph which fulfills the conditions of Lemma 2. Our algorithms need a data structure, called *path-access-structure*, that allows fast access to all paths of length at most k in H . Note that, by the properties of H from Lemma 2, the total length of all the paths is $O(n)$. We store the paths in an array S of length $O(n)$, which consists of padded consecutive subarrays S_v , for each node v of H . Each S_v contains each path starting in node v as consecutive cells. In order to access the paths we build up an array P which consists of padded consecutive subarrays P_v , 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 show how to build up the path-access-structure for paths of length at most k , and then how to update the structure under edge deletions.

Lemma 4. *The path-access-structure can be built on an n -processor DMM for all paths in H of length at most k w.h.p. in time $\mathcal{O}(\log k \log^* n)$, with linear total work.*

For the proof of Lemma 4 we need a lemma that was essentially proved in [CMS95].

Lemma 5. *In time $\mathcal{O}(\log^* n)$ and with linear total work, w.h.p., an n -processor DMM can compute the degree of each node of H and can store its adjacency list in a consecutive subarray of an array of length $O(n)$, evenly distributed among the modules.*

Proof of Lemma 4. The algorithm is based on the standard doubling technique (see e.g. [J92]). 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 . Additionally, if a given node has already found all simple paths then we call it *inactive*. Otherwise it is *active*. 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, Lemma 5 can be used to find the adjacency list in $\mathcal{O}(\log^* n)$ time with linear total work, w.h.p.. Additionally we inactivate all isolated nodes.

We perform the $(r + 1)$ -st iteration, for $r \geq 0$, by pointer jumping. Let v be any active node and C_v be its connected component. Note that since v is active $|C_v| \geq 2^r$. 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 least b 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 enables to compute $\tilde{\delta}_r(v)$ after performing the chaining algorithm. Now we compute approximately the total length, $\tilde{\gamma}_r(v)$, 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 path-length 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 their 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 $\tilde{\delta}_r(v) = O(|C_v|)$ and $\tilde{\gamma}_r(v) = O(|C_v| \cdot 2^r) = O(|C_v|^2)$. Hence the size of the new P_v is $O(|C_v|^2)$, and the size of the new S_v is $O(|C_v|^3)$. It is easy to compute the length of each new created path to maintain P_v . To update S_v we only have 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 the total work proportional to the sizes of the new P_v and S_v . This means that the total work for all nodes at all is $\sum_{C:|C|\geq 2^r} O(|C|^4)$, and the running time in each iteration is $\mathcal{O}(\log^* n)$, that is needed for computing $\tilde{\gamma}_r(v)$, $\tilde{\delta}_r(y)$ and allocate P_v 's and S_v 's.

Finally we have to remove the 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 easily verified, then the path p is not simple and we eliminate it. Strong semisorting within all arrays P_v can be used to remove non-simple paths from these arrays. Using the lengths of the paths in the P_v 's, approximate prefix sums enables to remove all non simple paths in the arrays S_v . Hence we can maintain the padded-consecutivity of the P_v 's and S_v 's. 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 total work $\mathcal{O}(\sum_{C:|C|\geq 2^r} |C|^4)$. Therefore, the total work of the algorithm is

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

and by Lemma 2, this is $\mathcal{O}(n)$.

Now we show how to maintain the path-access-structure when we allow removing edges from the graph. The proof of the following lemma can be obtained using techniques from the proof of Lemma 4.

Lemma 6. *Assume that the path-access-structure is already built. Then after removing some edges from the graph it can be updated in time $\mathcal{O}(\log^* n)$ with linear work on an n -processor DMM, w.h.p..*

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

Lemma 7. *Suppose that the path-access-structure is given. Then for all edges e and indices $r \in [k]$, the number of simple paths of length exactly r that start with edge e can be computed in $\mathcal{O}(\log^* n)$ time with linear work 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]. Now we perform strong semisorting with respect to these keys. Hence 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 $y_{e,r} \leq |X_{e,r}| \leq by_{e,r} = O(y_{e,r})$. 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 in constant time. Now we have to show that we only use $\mathcal{O}(n)$ processors. Let C_e be the connected component e belongs to. By Lemma 2, $y_{e,r} = O(|C_v|)$ and hence

$$\sum_{e \in E} \sum_{r=0}^k |X_{e,r}| \cdot 2^{|X_{e,r}|} \leq \sum_C \sum_{e \in C} |C| \cdot O(|C|) 2^{O(|C|)} \leq O\left(\sum_C |C|^3 2^{O(|C|)}\right)$$

Lemma 2 ensures that this is bounded by $O(n)$.

4 Triple-Logarithmic Simulation

One can view the one-out-of-two schedule as the following process on the graph H . Each processor that wants to access a shared memory cell $u \in U$ asks in each step either $M_{h_1(u)}$ or $M_{h_2(u)}$. This corresponds to directing the edge u in H to $M_{h_1(u)}$ or $M_{h_2(u)}$, respectively. Then, if a module M_j answers the request to cell u (that is, either the content of u is shown in the window of M_j during the reading phase, or the content of u is changed according to the processor that wants to write into u) then the edge labeled u is removed from H . That is, we direct every edge in H and then every node removes one edge (if any) that points to it. Before the next step starts, the orientations from the remaining edges are erased. The simulation ends when all the edges from H are removed.

The $\mathcal{O}(\log \log / \log \log \log n)$ -time simulation from [CMS95] 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 observation leading to improvements of that result is to look more globally and try to use information on as many nodes and edges as possible. The notion of the k -neighborhood plays the crucial role in our paper. 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 . 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 show in Section 3, essentially all information on the k -neighborhood can be computed in $\mathcal{O}(\log k \log^* n)$ time with linear total work. In this section we will

first show a process that removes all edges from a connected component with diameter δ in time $\mathcal{O}(\log \delta \log^* n)$. Then we present an algorithm that essentially breaks large connected components into smaller ones and then use the process on graphs with small diameter.

4.1 Cleaning up the Neighborhood

The access schedules described in previous papers (e.g. in [KLM92]) show how to remove all edges of a connected component C of H in time $\mathcal{O}(\log(|C|))$. The following lemma describes how to achieve a 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 12).

Lemma 8. (Cleaning up connected components)

Let δ be the maximal diameter of the connected components in H . Then one can remove all edges in H in time $\mathcal{O}(\log \delta \log^ n)$ with linear total work, w.h.p..*

Proof. As it is shown in Lemma 4, one can find, for each node v in H , all simple paths that start at v (and of course are of length at most δ) in time $\mathcal{O}(\log \delta \log^* n)$ with linear total work, w.h.p.. All these paths are stored in a padded-consecutive subarray S_v . Thus S_v contains exactly the nodes of v 's connected component. Now each node v can find the node w with the minimal identifier in its component. This can be easily done in $\mathcal{O}(\log^* n)$ time. If $v \neq w$, then v finds all nodes u_1, u_2, \dots, u_r , such that (v, u_i) is the first edge of a simple path from v to w . v "directs" the edges (v, u_i) , $1 \leq i \leq r$, to v , that is, the processor assigned to the edge (v, u_i) will try to access the module corresponding to the node v . Because of Lemma 2 we have $r = O(1)$, w.h.p.. Therefore, after $O(1)$ steps each processor will get the answer on its request.

4.2 An $\mathcal{O}(\log \log \log n \log^* n)$ -Time Simulation

In this section we describe a simulation of an n -processor EREW-PRAM on an n -processor DMM that improves all the previously known simulations exponentially. Essentially we show how to reduce the diameter of H to $(\log \log n)^2$ efficiently.

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 every node v has $\text{deg}(v)$ many k -branches. Define $\text{LEVEL}(r)$ of a k -branch of a node v to be the set of all simple paths of length r , $0 < r \leq k$, of this k -branch. The *weight* of a k -branch of a node v is the bit-vector $\overline{w} = (w_1, \dots, w_k)$. The value of w_r is 1 if and only if the number of simple paths in $\text{LEVEL}(r)$ of the branch is at least 2^{r-1} . If the weight of a k -branch satisfies $w_1 = w_2 = \dots = w_r = 1$, then we call it r -complete. 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.

SIMULATION

- *Each node v removes the incident edge which is the beginning of the k -branch with the maximal weight.*
- *Clean up all connected components.*

Using the path-access-structure, Lemma 4 and Lemma 7 we can easily derive an algorithm for the first step of SIMULATION, that needs time $\mathcal{O}(\log k \log^* n)$ and linear total work, w.h.p.. Now we prove that at the beginning of the clean up procedure the maximal diameter of each connected component in H is at most $O(k^2)$, w.h.p., for $k \geq \log \log n$.

Lemma 9. *Let $k \geq \log \log n$ and ζ denote the number of cycles in H . After the first step of the algorithm, w.h.p., H does not contain any simple path of length at least $(\zeta + 1) \cdot (2k + 1)^2$.*

Proof. Assume in the contrary, that at the beginning of the clean up procedure a simple path p of length $(\zeta + 1) \cdot (2k + 1)^2$ survives. Let us call each branch that starts with an edge from p the *path-branch* and we call any k -branch chosen in the first step of the algorithm by a node from p the *side-branch* of this node. Since no node 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 2 ensures that $\zeta = O(1)$. Hence there exists a subpath $\tilde{p} = (v_0, v_1, \dots, v_{2k})$ 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 $2k$.

We show that for each node from \tilde{p} the side-branch must be r -complete, for all nodes v_i , $0 \leq i \leq 2k - r$, the right path-branch (starting from the edge (v_i, v_{i+1})) is r -complete, and for all nodes v_i , $r \leq i \leq 2k$, 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 levels.

LEVEL(1) Because no node 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} . Since for each path-branch of a node from \tilde{p} we have $w_1 = 1$, all the desired path-branches and side-branches are 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 , $0 \leq i \leq 2k - r$, 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 they are 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 , $r \leq i \leq 2k$, can be treated in a similar way.

This implies that we need a connected structure of at least $2k \cdot 2^{k-1}$ nodes in H for a simple path of length $\zeta \cdot (2k + 1)^2$ to survive the first step of the algorithm. For $k \geq \log \log n$ this contradicts to Lemma 2.

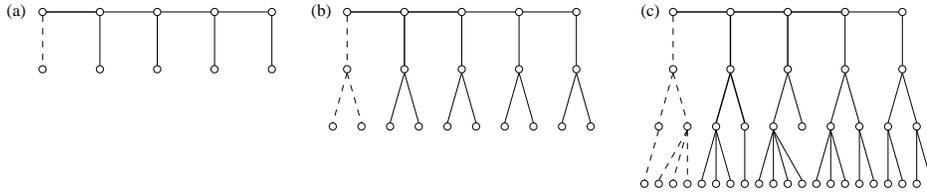


Fig. 1. (a) the path \tilde{p} with the edges that are removed by the nodes of \tilde{p} ; (b) the path \tilde{p} with the edges required by the elimination rule on LEVEL(2); (c) the path \tilde{p} with the edges required by the elimination rule on LEVEL(3).

The following theorem follows from Lemma 4, Lemma 7, Lemma 8, and Lemma 9.

Theorem 10. SIMULATION *simulates a step of an n -processor EREW PRAM on an n -processor DMM in $\mathcal{O}(\log \log \log n \log^* n)$ time with linear total work, w.h.p..*

5 A Lower Bound for PRAM Simulations

In this section we show a lower bound for shared memory simulations based on hashing and performing the *one-out-of-two schedule*. Again, we view the one-out-of-two schedule as a process of removing edges from the graph H as defined in Section 2. We consider the following class of *topological algorithms*:

- Each node of H knows its k -neighborhood; we charge for this $\mathcal{O}(\log k)$ time.
- All nodes repeat in parallel the following iteration until all edges are removed:
 - Each node that has degree one removes all edges in its k -neighborhood.
 - Each other node removes an incident edge basing its decision which one to remove only on the topology of its k -neighborhood.

Observe that this class of algorithms covers all previously known algorithms for randomized shared memory simulations (which essentially use at most the 1-neighborhood) and the algorithms presented in this paper. This also means that our simulation is almost optimal within the class of topological algorithms. The *cleaning up connected components* procedure (Lemma 8) is covered by the capability of the nodes of degree one to eliminate their whole k -neighborhood. In the proof of the lower bound we assume that we can without loss of generality delete edges from the graph H , i.e., with this operation we do not prejudice any

simulation. The capability of removing edges has positive effects for all known algorithms. Finally, a node can only get the knowledge of its k -neighborhood if the simulation needs time $\log k$. This maximal knowledge is given in the class of topological algorithms in advance.

The main idea of the lower bound is to focus only on completely symmetric structures in the access graph. Each node that has distance at least k from all leaves has a symmetric k -neighborhood. Hence it randomly makes the decision which outgoing edge to remove. We show that after performing these random decisions a smaller symmetric subgraph will still be left, with sufficiently high probability. The bound for the decrease of the size of the symmetric subgraph will yield the lower bound.

Definition 11. A (d_i, T_i) -tree is a complete d_i -ary tree of depth T_i .

Define the values of d_i and T_i for $0 \leq i < \frac{\log \log \log n}{8 \log \log \log \log n}$ as follows:

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

Fix an algorithm A that belongs to the class of topological algorithms. We want to maintain a (d_i, T_i) -tree in the access graph remaining after performing i iterations of the algorithm, with sufficiently high probability. The proof of this invariant is done by induction, which is based on the following two lemmas. Their proofs are omitted in this extended abstract. Let H be a random graph defined in Section 2.

Lemma 12. *Let $c < \log \log n$ and let T be a fixed tree with q nodes. For $q \leq \frac{\log n}{9 \log \log n}$ the probability that T is a subgraph of H is at least $1/2$.*

Lemma 13. *Consider a (d_i, T_i) -tree, for $0 \leq i < \frac{\log \log \log n}{8 \log \log \log \log n}$, $k \leq \sqrt{\log \log n}$. If every node randomly removes an incident edge then, with probability at least $1 - 1/\log n$, at most $d_i^{T_i - T_{i+1} - 2k}$ of the nodes have degree smaller than d_{i+1} .*

Using these two lemmas we can show that after iteration i of algorithm A a (d_i, T_i) -tree is left, with sufficiently high probability.

Lemma 14. *After performing i iterations of the algorithm A , a (d_i, T_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 12. Assume that the lemma holds for $i < \frac{\log \log \log n}{8 \log \log \log \log n}$. From the induction hypothesis we know that a (d_i, T_i) -tree is a subgraph of the access graph at the beginning of round $i + 1$. Without loss of generality, we only consider the edges from this tree and remove all other edges. Because of the definition of our class of topological algorithms we remove all nodes that have distance at most k from any leaf of this tree.

For the remaining nodes the topology of their k -neighborhood is fully symmetric, so it is not possible for them to distinguish between the incident edges because every permutation of the labeling of the edges and modules is equally likely. Therefore each decision based on the topology of the k -neighborhood made by these nodes is random and is independent on decisions of other nodes.

Hence, using Lemma 13 at most $d_i^{T_i - T_{i+1} - 2k}$ nodes have degree smaller than d_{i+1} , and a (d_i, T_i) -tree is a subgraph of the remaining graph.

The invariant of Lemma 14 holds in each iteration i , for $1 \leq i < \frac{\log \log \log n}{8 \log \log \log \log n}$, and k small enough, even if we start only with $\frac{n}{\log \log \log n}$ edges. This implies the following theorem.

Theorem 15. *For any algorithm of the class of topological algorithms the expected number of iterations until all edges of the access graph H will be removed is $\Omega(\frac{\log \log \log n}{\log \log \log \log n})$.*

Proof. In time $\mathcal{O}(\frac{\log \log \log n}{\log \log \log \log n})$ it is only possible to see a k -neighborhood for

$$k \leq 2^{O(\frac{\log \log \log n}{\log \log \log \log n})} \leq \sqrt{\log \log n}.$$

Therefore using Lemma 14, after $i \leq \frac{\log \log \log n}{8 \log \log \log \log n}$ iterations some edges will be left. The probability for this event can be bounded by

$$(1 - \frac{1}{\log n})^{\frac{\log \log \log n}{8 \log \log \log \log n}} \geq 1/e.$$

This yields the desired expected number of iterations.

References

- [BH93] H. Bast and T. Hagerup. Fast parallel space allocation, estimation and integer sorting (revised). Technical Report MPI-I-93-123, Max-Planck-Institut für Informatik, Im Stadtwald 66123 Saarbrücken, June 1993. A preliminary version appeared in *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing* under the title “Constant-Time Parallel Integer Sorting”, 1991.
- [BV93] O. Berkman and U. Vishkin. Recursive star-tree parallel data structure. *SIAM Journal on Computing*, 22(2):221–242, 1993. A preliminary version appeared in *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science*, pages 196–202, 1989.
- [CMS95] A. Czumaj, F. Meyer auf der Heide, and V. Stemann. Improved optimal shared memory simulations, and the power of reconfiguration. To appear in *Proceedings of the 3rd Israel Symposium on Theory of Computing and Systems*, 1995.
- [DM90] M. Dietzfelbinger and F. Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *Proceedings of the 17th Annual International Colloquium on Automata, Languages and Programming*, pages 6–19, 1990.

- [DM93] 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*, pages 110–119, 1993.
- [GMR94] 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*, pages 257–267, 1994.
- [GMV91] 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*, pages 698–710, 1991.
- [GMV94] 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*, pages 241–250, 1994.
- [J92] J. JáJá. *An Introduction to Parallel Algorithms*. Addison Wesley, 1992.
- [KLM92] R. M. Karp, M. Luby, and F. Meyer auf der Heide. Efficient PRAM simulation on a distributed memory machine. Technical Report tr-ri-93-134, University of Paderborn, 1993, to appear in *Algorithmica*. A preliminary version appeared in *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 318–326, 1992.
- [KU86] 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*, pages 160–168, 1986.
- [L92a] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann Publishers, San Mateo, 1992.
- [L92b] F. T. Leighton. Methods for packet routing in parallel machines. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 77–96, 1992.
- [M92] F. Meyer auf der Heide. Hashing strategies for simulating shared memory on distributed memory machines. In *Proceedings of the Heinz Nixdorf Symposium on Parallel Architectures and Their Efficient Use*, pages 20–29, 1992.
- [MPR94] P. D. MacKenzie, C. G. Plaxton, and R. Rajaraman. On contention resolution protocols and associated probabilistic phenomena. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, pages 153–162, 1994.
- [MSS95] F. Meyer auf der Heide, C. Scheideler, and V. Stemann. Exploiting storage redundancy to speed up randomized shared memory simulations. To appear in *Proceedings of the 12th Annual Symposium on Theoretical Aspects of Computer Science*, 1995.
- [R93] P. Ragde. The parallel simplicity of compaction and chaining. *Journal of Algorithms*, 14:371–380, 1993.
- [R91] A. G. Ranade. How to simulate shared memory. *Journal of Computer and System Sciences*, 42:307–326, 1991.
- [S89] 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*, pages 20–25, 1989.
- [U84] E. Upfal. Efficient schemes for parallel communication. *Journal of the ACM*, 31:507–517, 1984.
- [UW87] E. Upfal and A. Wigderson. How to share memory in a distributed system. *Journal of the ACM*, 34:116–127, 1987.

This article was processed using the L^AT_EX macro package with LLNCS style