ELSEVIER

Theoretical Computer Science 164 (1996) 107–122

Theoretical Computer Science

Reliable computations on faulty EREW PRAM

Krzysztof Diks [a,b,*], Andrzej Pelc [b,1]

[a] *Instytut Informatyki, Uniwersytet Warszawski, Banacha 2, 02-097 Warszawa, Poland*
[b] *Département d'Informatique, Université du Québec à Hull, Hull, Québec J8X 3X7, Canada*

Received July 1994; revised June 1995
Communicated by M. Crochemore

Abstract

We consider the problem of efficient and reliable computing on EREW PRAM whose processors are subject to random independent stop-failures with constant probability $p < 1$. An algorithm for such a fault-prone machine is called safe if it solves a problem of size n with probability exceeding $1 - d/n$, for some constant d independent of n. Our main contribution is a safe algorithm for the well-known list ranking problem, working in time $O(\log n)$ on an $O(n \log n)$-processor EREW PRAM. We also show an optimal safe algorithm for computing prefix sums, which works in time $O(\log n)$ on an $O(n/\log n)$-processor EREW PRAM. The methods presented in this paper can be applied to a wide class of EREW PRAM algorithms making them safe and simultaneously preserving their complexity.

1. Introduction

Important computation speed-ups permitted by massively parallel systems yield growing interest in efficient parallel algorithms. However, as the number of inexpensive general-purpose processors is increased to cope with problems of growing size, the number of potentially faulty processors grows accordingly. Parallel algorithms designed for fundamental computational problems in recent years (cf. [3, 5, 7, 9]) tend to use available processors very efficiently, leaving few of them idle at each step of execution. Such algorithms allow very restricted redundancy and consequently are usually fault sensitive: failures of even few processors can cause incorrect algorithm execution. This yields the need for parallel algorithms combining speed with reliability: the algorithm should be efficient and at the same time work correctly if a reasonable number of processors fail.

* Correspondence address: Institute of Informatics, Warsaw University, Banacha 2, 02-097 Warsaw, Poland. Email: diks@mimuw.edu.pl.
Research supported in part by NSERC International Fellowship and grant KBN.
[1] Research supported in part by NSERC grant OGP 0008136.

SSDI 0304-3975(95)00186-7

Recently, many authors have constructed fast and reliable parallel algorithms for important computaional problems, or even designed efficient and robust simulation techniques to transform any algorithm working in a fault-free environment into a reliable one working in a fault-prone system [8, 10, 11, 14, 15]. In all those papers the CRCW PRAM model of computation was adopted, while algorithms were supposed to work correctly even if only one processor remained fault-free. In [10, 11] a probabilistic fault model was used while efficiency criteria were expected execution time and expected work of proposed algorithms. It should be noted that algorithm complexities obtained in the above papers cannot be achieved with restricted concurrency, e.g. in the CREW PRAM or EREW PRAM models: concurrent write plays an essential role in providing necessary redundancy without increasing time.

In this paper reliable computations on a faulty EREW PRAM are studied for the first time. We assume that processors are subject to random independent stop-failures with constant probability $p < 1$. We seek algorithms which solve correctly a problem of size n with probability exceeding $1 - d/n$ for some constant d independent of n. Such algorithms are called *safe*. We consider two fundamental problems in parallel computations (cf. [5, 7]): *list ranking* (for a list given in array $S[1..n]$ where $S[i]$ is the number of the predecessor of i in the list, compute the distance of each i from the beginning of the list) and *prefix sums* (for a sequence of numbers $(a_1, \ldots, a_n)$ compute the sequence $(b_1, \ldots, b_n)$, where $b_i = a_1 + \cdots + a_i$). Our main contribution is a safe algorithm for the list ranking problem working in time $O(\log n)$ on an $O(n \log n)$-processor EREW PRAM. We simulate the well-known pointer jumping algorithm using expander graphs to schedule processors' allocation. (Our solution was inspired by the paper [2] of Assaf and Upfal where expanders are used to construct fault-tolerant sorting networks.) Then we apply this algorithm to get a safe solution to the prefix sums problem in time $O(\log n)$ on an $O(n/\log n)$-processor EREW PRAM. The methods presented in the paper can be applied to many other EREW PRAM algorithms (e.g. for expression evaluation, sorting, matrix multiplication) making them safe without increasing their running time and with the number of processors increased at most by logarithmic factor.

The paper is organized as follows. Section 2 contains precise model description and preliminary notions and facts used in the paper. In Section 3 we present a safe algorithm for the list ranking problem and in Section 4 we use it to solve the prefix sums problem in the presence of processor failures. Section 5 contains conclusions.

2. Model description and preliminaries

We work in the PRAM (Parallel Random Access Machine) model introduced by Fortune and Wyllie [4] and universally accepted as a model of synchronous parallel computations (cf. [3, 5, 7, 9]). In a PRAM, many processors each of which is a RAM, execute the same program in a synchronous way. Every processor has a positive integer identifier which can be used as a parameter of the program; hence, the actions

of different processors in program execution may differ, depending on their identifiers. Processors communicate through a shared memory from which they read data and to which they write results of local computations. In a unit of time a processor can access (read or write from/to) one memory cell. In this paper we use the most restrictive "Exclusive Read Exclusive Write" (EREW) variant of PRAM. In EREW PRAM different processors cannot attempt accessing the same memory cell at the same time.

We assume that processors may fail at each step of algorithm execution. Failures are independent, occur with probability $p < 1$ for each processor, and are of fail-stop type (cf. [8, 10, 11, 13–15]): a failed processor stops working and never restarts again. We assume that the action of writing to shared memory is atomic with respect to faults, i.e. a processor does not fail in the process of writing. Such fail-stop processor behavior with atomic computation steps is a realistic approximation of faults occurring in practice (cf. [13]). A PRAM in which processors are subject to random failures is called *unreliable*, a PRAM with all fault-free processors is called *ideal*.

Let A be an algorithm solving problem P of size n on a k-processor ideal EREW PRAM. The algorithm A is called *safe* if it solves P on a k-processor unreliable EREW PRAM M' with probability $R(A,n) > 1 - d/n$, where d is a constant independent of n. The probability $R(A,n)$ is called *reliability* of A (for size n).

In this paper all logarithms are taken with base 2. For an event E, $\bar{E}$ denotes its complement, while for a set X, $|X|$ denotes its size.

In our probabilistic considerations we use the following versions of Chernoff bound (cf. [6]).

Lemma 1. *Let S be the number of successes in a series of m Bernoulli trials with success probability q. Then*
(a) $\Pr(S \geqslant r) \leqslant 2^{-r}$, *for* $r \geqslant 6qm$;
(b) $\Pr(S \leqslant (1-\varepsilon)mq) \leqslant e^{-\varepsilon^2 mq/2}$, *for each* $0 < \varepsilon < 1$.

A bipartite graph $G = (A,B,E)$ is called an (α,β,m,d)-*expander* if $|A| = |B| = m$, the degree of each node of G is d and, for every set of nodes $X \subset A$ such that $|X| \leqslant \alpha m$, we have $|\Gamma(X)| \geqslant \beta|X|$, where $\Gamma(X)$ is the set of all neighbors of nodes from X. The following theorem was proved in [2] as a consequence of the construction given in [12].

Theorem 1. *For all $0 < \alpha < 1$ and $\beta > 0$ such that $\alpha\beta < 1$, there is an explicit construction of an (α,β,m,d)-expander with $d \leqslant 8\beta(1-\alpha)/(1-\alpha\beta)$.*

Corollary 1. *For every $0 < \lambda < \frac{1}{4}$ there exist constants α,β satisfying $2(\lambda+\alpha) < \alpha\beta < 1$ and such that an (α,β,m,d)-expander with $d \leqslant 8\beta(1-\alpha)/(1-\alpha\beta)$ can be explicitly constructed.*

Proof. Take a positive $\alpha < \frac{1}{4} - \lambda$ and $\beta = 2 + 1/2\alpha$. Then

$$2(\lambda+\alpha) < 2(\tfrac{1}{4}+\alpha) = \alpha\beta < 2(\tfrac{1}{4}+\tfrac{1}{4}-\lambda) < 1. \qquad \square$$

3. List ranking

The problem of list ranking is one of the fundamental problems appearing in the construction of efficient parallel algorithms (cf. [5, 7]). Consider a list L consisting of n nodes labeled by integers $1,\ldots,n$. Their order in the list is given in an array S such that $S[i]$ contains a pointer to the predecessor of i in the list, for all $i \leqslant n$, except the *first* element of the list, where $S[first] = 0$. The problem of list ranking consists in computing the distance of each node of the list from its beginning, i.e. computing an array R of integers, such that $R[first] = 0$ and $R[i] = R[S[i]] + 1$, for $i \neq first$. This problem can be solved optimally in time $O(\log n)$ on an $O(n/\log n)$-processor ideal EREW PRAM (cf. [1]).

In this section we present a safe algorithm for list ranking, working in time $O(\log n)$ on an $O(n \log n)$-processor unreliable EREW PRAM, for any probability $p < 1$ of processor failure. We will simulate the well-known pointer jumping algorithm. This algorithm is suboptimal: it works in time $O(\log n)$ on an n-processor ideal EREW PRAM (cf. [7]). We first recall the pointer jumping method in a version suitable for easy simulation.

The input is the array $S[1..n]$ and the output is the array $R[1..n]$, as defined above. We give the algorithm for processor p_i, $i = 1,\ldots,n$. Two auxiliary arrays $Q[1..n]$ and $T[1..n]$ will be used. Pointer jumping will be performed in Q, while $T[i]$ contains a time stamp indicating the last update concerning node i. Array T is not necessary for list ranking executed on an ideal PRAM but it will be used in the construction of a safe algorithm working in the presence of faults. For nodes i, j, $dist(i,j)$ denotes the distance in the list between nodes i and j.

Algorithm I (* List Ranking Using Pointer Jumping *)
(*stage 1: initialization *)
if $S[i] \neq 0$ **then**
$R[i] := 1; Q[i] := S[i]; T[i] := 1$
else
$R[i] := 0; Q[i] := i; T[i] := n$
fi;
(*stage 2: pointer jumping *)
for $step := 1$ **to** $\lceil \log n \rceil$ **do**
(* invariants:
inv1 - $T[i] \in \{step, n\}$;
inv2 - $T[i] = n \Rightarrow Q[i] = first$;
inv3 - $Q[i]$ precedes i in the list and $\mathrm{dist}(i, Q[i]) = R[i]$;
inv4 - $Q[i] \neq first \Rightarrow dist(i, Q[i]) = 2^{step-1}$. *)
$r_1 := R[i]$; $q_1 := Q[i]$; $t_1 := T[i]$;
if $t_1 = step$ **then**
$r_2 := R[q_1]$; $q_2 := Q[q_1]$; $t_2 := T[q_1]$;
$R[i] := r_1 + r_2$; $Q[i] := q_2$;

```
        if t2 = n then T[i] := n else T[i] := step + 1
      fi
  od
end (* of the algorithm *).
```

It is easy to see that inv1, inv2, inv3 and inv4 are indeed invariants of the **for** loop in stage 2. This implies

Lemma 2. *Algorithm* I *solves correctly the list ranking problem in time* $O(\log n)$ *on an n-processor ideal EREW PRAM.*

We now describe Algorithm II (Reliable List Ranking) which is a safe simulation of Algorithm I, working on unreliable EREW PRAM. First suppose that processor failure probability p is less than $\frac{1}{24}$. We will show later how this assumption can be dropped. Let $\lambda = 6p, c = 2/\lambda$ and $m = \lceil c \log n \rceil$. Algorithm II uses nm processors $p_{i,j}$, $i = 1,\ldots,n$, $j = 0,\ldots,m-1$. Let $P_i = \{p_{i,j} : 0 \leqslant j \leqslant m-1\}$. Processors from P_i collectively simulate actions of processor p_i in Algorithm I. The choice of m will guarantee that many processors in each P_i remain fault-free during the entire algorithm execution, with high probability. Thus, in the CRCW PRAM model it would suffice that every processor from P_i simply execute all actions of processor p_i in Algorithm I; however, in the EREW PRAM model simultaneous access to the same memory cell by many processors is forbidden, hence this simplistic idea does not work. Similarly, it would be easy to simulate Algorithm I on the EREW PRAM with a logarithmic slowdown; our algorithm, however, works on nm processors in time $O(\log n)$.

Algorithm II has the same input and output as Algorithm I. It uses two-dimensional arrays RR, QQ and TT (with n rows corresponding to nodes of the list and m columns numbered $0,\ldots,m-1$) which play the same role as their one-dimesional counterparts R, Q and T in Algorithm I. The main idea of our simulation is that after executing corresponding steps in both algorithms, many terms of the ith row of RR (resp. QQ and TT) be equal $R[i]$ (resp. $Q[i]$ and $T[i]$), even though some processors fail during the execution of Algorithm II.

Algorithm II works in three stages: *initialization*, *pointer jumping* and *result reporting*. The first two stages correspond exactly to the respective stages of Algorithm I. In stage 3 results of computations are transfered from array RR to the output array R.

We give the algorithm for processor $p_{i,j}, 1 \leqslant i \leqslant n, 0 \leqslant j \leqslant m-1$, from the set P_i.

Stage 1 – initialization

Arrays RR, QQ and TT are initialized so that elements of their ith rows are copies of $R[i], Q[i]$ and $T[i]$, respectively, after intialization in Algorithm I. The ith row in each array is initialized by processors from P_i in m steps. Every processor initializes each element of the row. In order to avoid access conflicts, this is done in a round-robin fashion. The following procedure is a formal description of this stage for processor $p_{i,j}$.

```
procedure initialize(i,j);
(* preprocessing *)
   for k := 0 to m − 1 do
      if j = k then q := S[i];
   if q = 0 then (* i = first *)
            t := n; r := 0
      else
            t := 1; r := 1
   fi;
(* proper initialization *)
   for k := 0 to m − 1 do
      RR[i,(j + k) mod m] := r;
      QQ[i,(j + k) mod m] := q;
      TT[i,(j + k) mod m] := t
   od
end (* of the procedure *).
```

It is easy to see that if at least one processor in each set P_i remains fault-free after execution of the above procedure then, for all $1 \leqslant i \leqslant n$ and $0 \leqslant j \leqslant m-1$, $RR[i,j] = R[i]$, $QQ[i,j] = Q[i]$ and $TT[i,j] = T[i]$ after initializations in Algorithms I and II.

Stage 2 – pointer jumping

This is a simulation of Stage 2 of Algorithm I. In every execution of its **for** loop, processor p_i reads at most two triples of data: (r_1, q_1, t_1) from $R[i]$, $Q[i]$ and $T[i]$, respectively, and (r_2, q_2, t_2) from $R[q_1]$, $Q[q_1]$ and $T[q_1]$, respectively. If necessary, processor p_i writes new data $(r_1 + r_2, q_2, t)$ in $R[i]$, $Q[i]$ and $T[i]$, respectively.

In Algorithm II, every processor $p_{i,j} \in P_i$ acts similarly but instead of reading one triple of data from rows i of RR, QQ and TT and one triple of data from rows q_1 of these arrays, it reads a constant number of triples from rows i and q_1, and accepts those with the largest time stamp t (most recently updated). If accepted triples have the same time stamps as corresponding triples in Algorithm I, processor $p_{i,j}$ performs the same computations as p_i in Algorithm I and writes their results in $RR[i,j], QQ[i,j]$ and $TT[i,j]$. The choice of triples read by $p_{i,j}$ is given by a fixed (α, β, m, d)-expander $G = (A, B, E)$ (where $A = \{x_0, \ldots, x_{m-1}\}, B = \{y_0, \ldots, y_{m-1}\}$), such that α and β satisfy assumptions of Corollary 1. Let $E_1, \ldots, E_d$ be a partition of all links from E into d perfect matchings. For all $0 \leqslant j \leqslant m-1, 1 \leqslant k \leqslant d$, let $a(j,k)$ be such that $(x_j, y_{a(j,k)}) \in E_k$. Without loss of generality assume $a(j,1) = j$ for every $j = 0, \ldots, m-1$. If processor $p_{i,j}$ has to read triples of data from sth rows of arrays RR, QQ and TT, it reads consecutive terms $a(j,1), \ldots, a(j,d)$ of these rows and keeps the triple with highest time stamp t. Since indices $a(j,1), \ldots, a(j,d)$ are given by matchings, access conflicts are avoided. The following procedure formalizes the action of getting data by processor $p_{i,j}$ from rows s of the respective arrays:

```
procedure get_data(i, j; s; r, q, t);
  t := 0;
  for k := 1 to d do
      if TT[s, a(j,k)] > t then
        r := RR[s, a(j,k)];
        q := QQ[s, a(j,k)];
        t := TT[s, a(j,k)]
  fi
end (* of the procedure *).
```

The simulation of p_i's behavior by processor $p_{i,j}$ is performed by the following procedure:

```
procedure rank(i, j);
  for step := 1 to ⌈log n⌉ do
 (* invariant INV: for all 1 ⩽ u ⩽ n, 0 ⩽ v ⩽ m − 1,
TT[u, v] ⩾ step ⇒ TT[u, v] = T[u], RR[u, v] = R[u], QQ[u, v] = Q[u],
where values of T, R, Q are taken just before execution number step
of the for loop in stage 2 of Algorithm I. *)
      get_data(i, j; i; r1, q1, t1);
      if t1 = step then
        get_data(i, j; q1; r2, q2, t2);
        if t2 ⩾ step then
          QQ[i, j] := q2; RR[i, j] := r1 + r2;
          if t2 = n then TT[i, j] := n else TT[i, j] := step + 1
        fi
      fi
  od
end (* of the procedure *).
```

The following example shows the execution of a particular iteration of the loop in procedure rank(i, j).

Example. Suppose that the length n of the list is 1024, $m = 30$, the vertex degree d of the expander G is 5, and the function a is such that $a(10,1) = 10$, $a(10,2) = 15$, $a(10,3) = 7$, $a(10,4) = 25$, $a(10,5) = 18$. Consider node 100 on the list with distance from the head larger than 32 and assume that processor $p_{100,10}$ does not fail during the first five iterations of the procedure rank. Suppose that before the fifth iteration

TT[100, 10] = **5**, $TT[100,15] = 3$, **TT[100, 7]** = **5**,
TT[100, 25] = **5**, $TT[100,18] = 1$;
QQ[100, 10] = **500**, $QQ[100,15] = 70$, **QQ[100, 7]** = **500**,
QQ[100, 25] = **500**, $QQ[100,18] = 200$;
RR[100, 10] = **16**, $RR[100,15] = 8$, **RR[100, 7]** = **16**, **RR[100, 25]** = **16**,
$RR[100,18] = 1$;
$TT[500,10] = 2$, **TT[500, 15]** = **5**, $TT[500,7] = 1$, **TT[500, 25]** = **5**,
TT[500, 18] = **5**;
$QQ[500,10] = 30$, **QQ[500, 15]** = **40**, $QQ[500,7] = 600$,
QQ[500, 25] = **40**, **QQ[500, 18]** = **40**;
$RR[500,10] = 4$, **RR[500, 15]** = **16**, $RR[500,7] = 2$, **RR[500, 25]** = **16**,
RR[500, 18] = **16**.

(Up to date data are bold faced.) During the fifth iteration processor $p_{100,10}$ reads five triples from columns 10, 15, 7, 25, 18 in rows with index 100 of arrays QQ, RR, TT and stores the triple with the largest TT, i.e. $(q_1 = 500, r_1 = 16, t_1 = 5)$. Since $t_1 = 5$, processor $p_{100,10}$ reads the next five triples from columns 10, 15, 7, 25, 18 in rows with index 500 of arrays QQ, RR, TT and again it accepts the one with the largest TT, i.e. $q_2 = 40$, $r_2 = 16, t_2 = 5$. Finally, it sets $QQ[100,10] = 40, RR[100,10] = 32, TT[100,10] = 6$.

Stage 3 – result reporting

Correct results from array RR, those for which the corresponding value TT is n, are copied into the output array R. Stage 3 is executed in a round-robin fashion, similarly as Stage 1, using the following procedure:

```
procedure final_result(i, j);
  for k := 0 to m − 1 do
      if TT[i, (j + k) mod m] = n then
        r := RR[i, (j + k) mod m];
  for k := 0 to m − 1 do
      if j = k then
        R[i] := r
end (* of the procedure *).
```

Now the entire algorithm for processor $p_{i,j}$ can be formulated as follows:

```
Algorithm II (* Reliable List Ranking *)
initialize(i, j);
rank(i, j);
final_result(i, j)
end (* of the algorithm *).
```

Procedure initialize works in time $O(m)$, procedure rank works in time $O(\log n)$ and procedure final_result works in time $O(m)$. Hence, Algorithm II works in time $O(\log n)$. It uses $nm \in O(n \log n)$ processors.

We now prove that Algorithm II is a safe solution of the list ranking problem.

Lemma 3. *Let E be the event that every set P_i, for $i \leqslant n$, contains at least $(1-\lambda)m$ processors which remain fault-free during the execution of Algorithm II. Then $\Pr(E) \geqslant 1 - 1/n$, for sufficiently large n.*

Proof. Lemma 1 implies

$$\Pr(\bar{E}) \leqslant n2^{-6pm} \leqslant n2^{-6pc \log n} = \frac{1}{n}. \qquad \square$$

In all further considerations in this section we assume that E holds. Thus, after initializations in Algorithms I and II we have

$$TT[i,j] = T[i], \quad RR[i,j] = R[i], \quad QQ[i,j] = Q[i]$$

for all $1 \leqslant i \leqslant n$ and $0 \leqslant j \leqslant m-1$.

We now prove that INV is indeed an invariant of the **for** loop in procedure rank. For $s = 1, \ldots, \lceil \log n \rceil + 1$ let $TT^s[u,v]$ $(RR^s[u,v], QQ^s[u,v])$ be values of $TT[u,v]$ $(RR[u,v], QQ[u,v])$ before the sth execution of the **for** loop in the procedure rank (if $s = \lceil \log n \rceil + 1$, we consider the value after the last execution of the loop). $T^s[u], R^s[u]$ and $Q^s[u]$ are defined similarly, with respect to the **for** loop in Algorithm I.

Lemma 4. *For every $s = 1, \ldots, \lceil \log n \rceil + 1$ and all $1 \leqslant u \leqslant n, 0 \leqslant v \leqslant m-1$ the following holds:*

$$TT^s[u,v] \geqslant s \Rightarrow TT^s[u,v] = T^s[u], \quad RR^s[u,v] = R^s[u], \quad QQ^s[u,v] = Q^s[u]. \qquad (*)$$

Proof. Induction on s:

Since event E holds, $(*)$ is satisfied for $s = 1$ (after initialization). Assume that it is satisfied for some s; we will prove that it is satisfied for $s+1$ (after the sth execution of the loop). Fix a pair (u,v). During the sth execution of the loop values of $QQ[u,v], RR[u,v]$ and $TT[u,v]$ can be changed only by processor $p_{u,v}$ and they are changed in this order: the time stamp is changed last. If $p_{u,v}$ fails before executing the assignment for $TT[u,v]$ in the sth execution of the loop then either $TT^s[u,v] \leqslant s$ and $TT^{s+1}[u,v] < s+1$ or $TT^s[u,v] = TT^{s+1}[u,v] = n$. In the first case, $(*)$ holds for $s+1$ because the assumption is not satisfied; in the second case no values are changed in both algorithms:

$$QQ^{s+1}[u,v] = QQ^s[u,v] = Q^s[u] = Q^{s+1}[u],$$

$$RR^{s+1}[u,v] = RR^s[u,v] = R^s[u] = R^{s+1}[u],$$

hence $(*)$ holds as well. Thus, we may assume that $p_{u,v}$ remains fault-free during the entire sth execution of the loop.

First, $p_{u,v}$ gets the triple (r_1, q_1, t_1). Consider three cases:

1. $t_1 < s$: In this case the data got by $p_{u,v}$ are not up to date. In particular, $TT^s[u,v] < s$. Thus, $p_{u,v}$ does not change the value of $TT[u,v]$ in this execution of the loop and consequently $(*)$ remains true for $s+1$.

2. $t_1 = s$: It follows from the description of Algorithm II that $TT^s[u,v] \leqslant s$. Since $t_1 = s$, $p_{u,v}$ gets the second triple (r_2, q_2, t_2) and behaves differently depending on the value of t_2:

(a) $t_2 < s$. The values read by $p_{u,v}$ are not up to date. Processor $p_{u,v}$ does not change values of $TT[u,v], RR[u,v]$ and $QQ[u,v]$ and $(*)$ remains true for $s+1$.

(b) $t_2 \geqslant s$. In view of $(*)$ for s and of the description of Algorithm II, $p_{u,v}$ modifies the triple $(QQ[u,v], RR[u,v], TT[u,v])$ in the same way as p_u modifies the triple $(Q[u], R[u], T[u])$. Hence $(*)$ remains true for $s+1$.

3. $t_1 > s$: In view of $(*)$ for s and of the description of Algorithm II we get $t_1 = n$ and either $TT^s[u,v] < s$ or $TT^s[u,v] = n$. Processor $p_{u,v}$ does not change the value of $TT[u,v]$. If $TT^s[u,v] < s$ then $TT^{s+1}[u,v] < s+1$ and hence values of $QQ[u,v]$ and $RR[u,v]$ remain unchanged. If $TT^s[u,v] = n$ then $TT^{s+1}[u,v] = n = T^{s+1}[u]$ and

$$QQ^{s+1}[u,v] = QQ^s[u,v] = Q^s[u] = Q^{s+1}[u],$$

$$RR^{s+1}[u,v] = RR^s[u,v] = R^s[u] = R^{s+1}[u].$$

Hence, also in this case, $(*)$ holds for $s+1$. □

Lemma 5. *There are no conflicts of access to the shared memory in the execution of Algorithm II.*

Proof. It is easy to see that access conflicts do not arise in Stages 1 and 3. In Stage 2 processors from P_i read data from the uth rows of arrays RR, QQ, TT in a given step of Algorithm II iff processor p_i reads data from $R[u], Q[u], T[u]$ in the corresponding step of Algorithm I. Since Algorithm I does not yield access conflicts, it follows that processors from distinct sets P_i, P_j do not read simultaneously data from the same row of arrays RR, QQ, TT. Hence, conflicts could only arise among processors from the same set P_i. However, column numbers of memory cells from which processors from P_i read in a given step, are determined by matchings in the expander G, thus precluding any conflicts of access to the shared memory. □

The following is the key lemma of our proof. It implies that in every row of arrays RR, QQ and TT a fixed fraction of entries remain up to date during the entire execution of Algorithm II. In the proof of this lemma expander properties are used.

Lemma 6. *For every* $1 \leqslant s \leqslant \lceil \log n \rceil + 1$ *and every* $1 \leqslant i \leqslant n$,

$$|\{j : 0 \leqslant j \leqslant m-1, TT^s[i,j] \geqslant s\}| \geqslant (1 - \alpha - \lambda)m. \qquad (**)$$

Proof. Induction on s.

In every set P_i at least $(1-\lambda)m$ processors remain fault-free after initialization. Thus,

$$|\{j : 0 \leqslant j \leqslant m-1, TT^1[i,j] \geqslant 1\}| = m$$

for all $i \leqslant n$. Suppose that $(**)$ holds before sth execution of the **for** loop in Stage 2 of Algorithm II. We show that this condition remains true after the sth execution.

Consider the set P_i. At most λm processors from this set fail during algorithm execution. Every fault-free processor reads data from d distinct cells of ith rows of arrays RR, QQ and TT. By the inductive assumption at least $(1-\alpha-\lambda)m$ entries of the ith row of TT are $\geqslant s$. Suppose that at least $\alpha m/2$ processors read all d triples of data with time stamp (the entry in TT) smaller than s. Let X be the set of $\lceil \alpha m/2 \rceil$ such processors. Since G is an (α, β, m, d)-expander, it follows that processors from X read values from at least $\beta|X|$ distinct positions of the ith row of TT. Hence, at least $\beta|X|$ entries of the ith row of TT are $< s$. However,

$$\beta|X| = \beta \left\lceil \frac{\alpha m}{2} \right\rceil > (\lambda + \alpha)m,$$

which yields a contradiction.

Let Y be the set of fault-free processors which read an up to date triple (r_1, q_1, t_1) (i.e. such that $t_1 \geqslant s$) during sth execution of the loop. The above contradiction implies that $|Y| \geqslant (1-\lambda-\alpha/2)m$. Every processor in Y simulates actions of processor p_i in Algorithm I. In particular, if $t_1 = s$, it reads d triples (r_2, q_2, t_2) and keeps the triple with the largest time stamp t_2. Using the same argument as before we can show that all processors from Y, except less than $\alpha m/2$, obtain an up to date triple. Hence, at least $(1-\lambda-\alpha)m$ processors from P_i get both up to date triples in the sth execution of the loop and consequently perform correct computations and write results in respective arrays. In particular, $(**)$ holds for $s+1$, which proves the lemma by induction. □

Lemmas 2–6 imply

Lemma 7. *Algorithm* II *is a safe algorithm for list ranking on an unreliable EREW PRAM, for processor failure probability* $p < \frac{1}{24}$.

It remains to generalize the above result for arbitrary $p < 1$. Let k be the minimum integer for which $p^k < \frac{1}{24}$. Replace every processor π in Algorithm II by a set Π of k processors which sequentially repeat every action of π. The probabilty that all processors in Π fail during algorithm execution is smaller than $\frac{1}{24}$ and our previous analysis can be applied. This yields the main result of this section.

Theorem 2. *Safe list ranking of an n-element list can be done in time* $O(\log n)$ *on an* $O(n \log n)$*-processor unreliable EREW PRAM.*

Unfortunately, we are not able to apply our techniques to the optimal list ranking algorithm for ideal EREW PRAM (working in time $O(\log n)$ on $O(n/\log n)$ processors,

cf. [1, 7]) and obtain in this way a safe algorithm running in logarithmic time on $O(n)$ processors of unreliable EREW PRAM. In the pointer jumping method each processor is responsible for updating a portion of data fixed in advance, i.e. processor p_i updates only $R[i]$, $Q[i]$ and $T[i]$. This allows to set time stamps $T[i,j]$ in such a way that every processor reading a triple $R[i,j]$, $Q[i,j]$, $T[i,j]$ can easily detect whether information is up to date. However, in the optimal list ranking algorithm data corresponding to every node of the list can be updated many times and by different processors, in stages of the algorithm execution which are not fixed in advance. This is the main reason why we are not able to build a mechanism allowing processors of unreliable EREW PRAM to detect whether obtained data are correct. On the other hand, actions performed on old versions of data could cause memory conflicts and incorrect termination of algorithm execution.

4. Prefix sums

In this section we apply our safe list ranking algorithm to get a safe algorithm computing prefix sums of an n-element sequence in time $O(\log n)$ on an $O(n/\log n)$-processor unreliable EREW PRAM. Thus, as opposed to the list ranking algorithm from the previous section, our algorithm to compute prefix sums is optimal, i.e. the product (time $\times$ number of processors) is of the same order of magnitude $O(n)$ as the time of the best sequential algorithm.

The Prefix Sums Problem can be formulated as follows: given an array of numbers $A[1..n]$ compute the array $R[1..n]$, where $R[i] = A[1] + \cdots + A[i]$, for all $i \leqslant n$.

First note that the above problem can be easily solved by a straightforward application of a list ranking algorithm to the list $S[i] = i - 1$, for all $i = 1,\ldots,n$. The only modification needed (call it modification $(*)$) is in the initialization stage: for all $1 \leqslant i \leqslant n$, $R[i]$ should be initialized as $A[i]$ (instead of 1, or 0 for the first element). Thus, using Theorem 2, we immediately get a safe algorithm computing prefix sums in time $O(\log n)$ on an $O(n \log n)$-processor unreliable EREW PRAM. The aim of this section is to show how applying list ranking to sequences of length $O(n/\log^2 n)$ can reduce the number of processors required for the original problem by a factor $\Theta(\log^2 n)$.

For simplicity of further presentation we assume that n is such that $\log n$ is an integer dividing n. Consider the following auxiliary problem: Let $k = n/\log n$. Given an array $A[1..n]$, compute the array $B[1..k]$, such that $B[i] = \sum_{j=(i-1)\log n+1}^{i\log n} A[j]$. Notice that, for all $1 \leqslant i \leqslant k$ and $1 \leqslant m \leqslant \log n$,

$$R[(i-1)\log n + m] = \sum_{j=1}^{i-1} B[j] + \sum_{j=1}^{m} A[(i-1)\log n + j].$$

We now present the algorithm REDUCTION which safely computes the array B in time $O(\log n)$ on a k-processor unreliable EREW PRAM. We describe the algorithm informally leaving easy details to the reader. Divide the array A into k segments $A_i =$

$A[(i-1)\log n + 1..i\log n]$, for $i = 1,\ldots,k$. Let $p_0,\ldots,p_{k-1}$ be the processors of the PRAM. During the algorithm execution every segment A_i is visited by $c\log n$ distinct processors $p_{i-1}, p_{i \bmod k}, p_{(i+1) \bmod k},\ldots$, with the constant c chosen so as to guarantee that every A_i be visited by at least $\log n$ processors remaining fault-free until the end. For every $i \leqslant k$ consider a partial sum s_i and index $last_i$ such that $s_i = \sum_{j=1}^{last_i} A[(i-1)\log n + j]$. The role of a processor visiting the segment A_i is adding $A[(i-1)\log n + last_i + 1]$ to s_i (if $last_i < \log n$) and then incrementing $last_i$ by 1. Since at least $\log n$ fault-free processors visit A_i, s_i should be equal $B[i]$ upon completion of the algorithm.

There is one difficulty, however. It is possible that after modifying s_i, but before modifying $last_i$, a processor visiting A_i fails. Then the next fault-free processor visiting A_i would add again the element $A[(i-1)\log n + last_i + 1]$ to s_i (with unchanged $last_i$), thus producing a possibly incorrect result. In order to avoid this situation we keep two pairs $(s_i^0, last_i^0)$ and $(s_i^1, last_i^1)$. The pair $(s_i^\varepsilon, last_i^\varepsilon)$ is *correct* if

$$\sum_{j=1}^{last_i^\varepsilon} A[(i-1)\log n + j] = s_i^\varepsilon.$$

We ensure the invariant that, among pairs $(s_i^0, last_i^0), (s_i^1, last_i^1)$, the one with larger $last_i^\varepsilon$ is correct. A processor visiting A_i takes the pair $(s_i^\varepsilon, last_i^\varepsilon)$ such that $last_i^\varepsilon > last_i^{1-\varepsilon}$ and makes two assignments:

$$s_i^{1-\varepsilon} := s_i^\varepsilon + A[(i-1)\log n + last_i^\varepsilon + 1];$$

$$last_i^{1-\varepsilon} := last_i^\varepsilon + 1.$$

If the processor fails right after the first assignment, the inequality $last_i^\varepsilon > last_i^{1-\varepsilon}$ remains true and the next fault-free processor visiting α_i uses the correct pair $(s_i^\varepsilon, last_i^\varepsilon)$ for further computations, thus guaranteeing their correctness. If, on the other hand, the processor survives both assignments, we have

$$last_i^{1-\varepsilon} = last_i^\varepsilon + 1 > last_i^\varepsilon$$

and the pair $(s_i^{1-\varepsilon}, last_i^{1-\varepsilon})$ is indeed correct, which ensures the invariant.

Lemma 8. *Algorithm REDUCTION works in time* $O(\log n)$ *on a k-processor EREW PRAM and it is safe for a sufficiently large constant c.*

Proof. The first part of the lemma is obvious. On the other hand, if every segment A_i is visited by at least $\log n$ processors which remain fault-free during the algorithm execution then the output array B is correctly computed.

Let E_i be the event that among $c\log n$ processors visiting A_i, at least $\log n$ are fault-free. Lemma 1(b) with $q = 1 - p$ and $\varepsilon = (cq-1)/cq$, implies

$$\Pr(\bar{E}_i) \leqslant e^{-(cq-2+(1/cq))(\log n - 1)/2} \leqslant e^{(-cq\log n)/8} \leqslant \frac{1}{n^2}$$

for sufficiently large c and n. Thus,

$$\Pr\left(\bigcup_{i=1}^{k}\bar{E}_i\right) \leqslant \frac{n}{\log n}\frac{1}{n^2} < \frac{1}{n},$$

which implies that the algorithm is safe. □

We now show how to reduce the problem of computing prefix sums of array A to that of computing prefix sums of array B. Suppose the latter problem is solved and let the prefix sums of B be contained in array $B'[1..k]$ such that $B'[i] = \sum_{j=1}^{i} B[j]$, for $1 \leqslant i \leqslant k$. Observe that

$$R[(i-1)\log n + m] = B'[i-1] + \sum_{j=1}^{m} A[(i-1)\log n + j].$$

It follows that given the array B' we can safely compute the array $R[1..n]$ (containing prefix sums of sequence $A[1..n]$) using the same ideas as those in the algorithm REDUCTION. We will call this new algorithm REDUCTION'.

We finally describe the main algorithm of this section. Its input is a sequence of numbers given in array $A[1..n]$. Its output is the array $R[1..n]$ such that $R[j] = \sum_{m=1}^{j} A[m]$. For simplicity we assume that $\log^2 n$ is an integer dividing n. Modifications in the general case are easy.

Algorithm Prefix-Sums

1. Compute array $B_1[1..n/\log n]$ such that $B_1[i] = \sum_{j=(i-1)\log n+1}^{i\log n} A[j]$, using algorithm REDUCTION.
2. Compute array $B_2[1..n/\log^2 n]$ such that $B_2[i] = \sum_{j=(i-1)\log n+1}^{i\log n} B_1[j]$, using algorithm REDUCTION.
3. Compute array $R_2[1..n/\log^2 n]$ containing prefix sums of array B_2 (i.e. such that $R_2[i] = \sum_{j=1}^{i} B_2[j]$), using algorithm Reliable List Ranking with modification (*).
4. Compute array $R_1[1..n/\log n]$ containing prefix sums of array B_1 (i.e. such that $R_1[i] = \sum_{j=1}^{i} B_1[j]$), using algorithm REDUCTION' and using array R_2 from step 3.
5. Compute array $R[1..n]$ containing prefix sums of array A (i.e. such that $R[i] = \sum_{j=1}^{i} A[j]$), using algorithm REDUCTION' and using array R_1 from step 4.

end (* of the algorithm *).

Theorem 9. *Algorithm Prefix-Sums is a safe algorithm to compute prefix sums of an n-element sequence in time* $O(\log n)$ *on an* $O(n/\log n)$*-processor unreliable EREW PRAM.*

Proof. Each of the steps takes time $O(\log n)$. Steps 1 and 5 require $O(n/\log n)$ processors, steps 2 and 4 require $O(n/\log^2 n)$ and step 3 requires $O(n/\log n)$ processors

(Reliable List Ranking is applied to a list of length $O(n/\log^2 n)$). Thus, Algorithm Prefix-Sums works in time $O(\log n)$ on an $O(n/\log n)$-processor unreliable EREW PRAM. Since all algorithms: Reliable List Ranking, REDUCTION, REDUCTION' are safe, Prefix-Sums is safe as well. □

5. Conclusion

We presented reliable and efficient algorithms to solve two important computational problems on an unreliable EREW PRAM. This is the first time that this restricted computation model is more deeply studied from the point of view of fault tolerance. The main contribution of this paper is a simulation technique applied to the pointer jumping algorithm for list ranking, which permits to transform it into a safe algorithm in the presence of random processor failures. The actions of every processor in the original algorithm are simulated by $O(\log n)$ processors, thus multiplying the number of required processors by a logarithmic factor. The techniques introduced in this paper can be applied to some other algorithms, e.g. binary tree contraction (cf. [7]) if leaves of the tree are given in order left to right, thus yielding a safe algorithm for this problem, working in time $O(\log n)$ on an unreliable $O(n/\log n)$-processor EREW PRAM. First the size of the problem can be reduced to $O(n/\log^2 n)$, similarly as we did for the prefix sums problem, and then tree contraction can be simulated on the smaller tree associating $O(\log n)$ processors with every node.

We do not have, however, a general method to transform algorithms working on ideal EREW PRAM into safe algorithms working in the presence of random faults, with constant slowdown and increase of the number of processors at most by logarithmic factor. Such general methods were presented in [10] for CRCW PRAM and obtaining their counterparts for EREW PRAM remains the main open problem naturally suggested by our work. In particular, it remains open if safe list ranking can be done in time $O(\log n)$ on an $O(n)$-processor unreliable EREW PRAM. If our techniques could be applied to the optimal list ranking algorithm for ideal EREW PRAM (working in time $O(\log n)$ on $O(n/\log n)$ processors, cf. [1]) the answer to the above problem would be positive.

References

[1] R. Anderson and G. Miller, Deterministic parallel list ranking, in: *VLSI Algorithms and Architectures, Proc. 3rd Aegean Workshop on Computing*, Lecture Notes in Computer Science, Vol. 319 (Springer, Berlin, 1988) 81–90.

[2] S. Assaf and E. Upfal, Fault-tolerant sorting networks, *SIAM J. Discrete Math.* **4** (1991) 472–480.

[3] D. Eppstein and Z. Galil, Parallel techniques for combinatorial computations, *Ann. Comput. Sci. Rev.* **3** (1988) 233–283.

[4] S. Fortune and J. Wyllie, Parallelism in random access machines, *Proc. 10th ACM Symp. on Theory of Computing* (1978) 114–118.

[5] A.M. Gibbons and W. Rytter, *Efficient Parallel Algorithms* (Cambridge Univ. Press, Cambridge, 1988).

[6] T. Hagerup and C. Rüb, A guided tour of Chernoff bounds, *Inf. Proc. Lett.* **33** (1989/90) 305–308.
[7] J. JaJa, *An Introduction to Parallel Algorithms* (Addison-Wesley, Reading, MA, 1992).
[8] P.C. Kanellakis and A.A. Shvartsman, Efficient parallel algorithms can be made robust, Distributed Comput. **5** (1992) 201–217.
[9] R.M. Karp and V. Ramachandran, Parallel algorithms for shared memory machines, in: J. van Leeuven, ed., *Handbook of Theoretical Computer Science*, Vol A: *Algorithms and Complexity* (Elsevier, Amsterdam, 1990) 869–942.
[10] Z.M. Kedem, K.V. Palem, A. Raghunathan and P. Spirakis, Combining tentative and definite executions for very fast dependable parallel computing, in: *Proc. 23rd ACM Symp. on Theory of Computing* (1991) 381–390.
[11] Z.M. Kedem, K.V. Palem, P. Spirakis, Efficient robust parallel computations, in: *Proc. 22nd ACM Symp. on Theory of Computing* (1990) 138–148.
[12] A. Lubotzky, R. Philips and P. Sarnak, Explicit expanders and the Ramanujan conjectures, in: *Proc. 18th Ann. Symp. on Theory of Computing* (1986) 240–246.
[13] R.D. Schlichting and F.B. Schneider, Fail-stop processors: an approach to designing fault-tolerant computing systems, *ACM Trans. Comput. Systems* **1** (1983) 222–238.
[14] A.A. Shvartsman, Achieving optimal CRCW PRAM fault-tolerance, *Inform. Process. Lett.* **39** (1991) 59–66.
[15] A.A. Shvartsman, An efficient write-all algorithm for fail-stop PRAM without initialized memory, *Inform. Process. Lett.* **44** (1992) 223–231.