PRAM Computations Resilient to Memory Faults*

B.S. Chlebus A. Gambin P. Indyk

Instytut Informatyki, Uniwersytet Warszawski, Banacha 2, 02-097 Warszawa, Poland.
E-mail: chlebus@mimuw.edu.pl, aniag@zaa.mimuw.edu.pl, indyk@mimuw.edu.pl

Abstract: PRAMs with faults in their shared memory are investigated. Efficient general simulations on such machines of algorithms designed for fully reliable PRAMs are developed.

The PRAM we work with is the Concurrent-Read Concurrent-Write (CRCW) variant. Two possible settings for error occurrence are considered: the errors may be either static (once a memory cell is checked to be operational it remains so during the computation) or dynamic (a potentially faulty cell may crash at any time, the total number of such cells being bounded). A simulation consists of two phases: memory formatting and the proper part done in a step-by-step way. For each error setting (static or dynamic), two simulations are presented: one with a $O(1)$-time per-step cost, the other with a $O(\log n)$-time per-step cost. The other parameters of these simulations (number of processors, memory size, formatting time) are shown in table 1 in section 6. The simulations are randomized and Monte Carlo: they always operate within the given time bounds, and are guaranteed to be correct with a large probability.

1 Introduction

Parallel Random Access Machine (PRAM) is a popular model to design parallel algorithms (see [10, 12]). It is a multiprocessor system in which every processor acts like a RAM, and all of them share the global memory. PRAM abstracts from real multiprocessor computers by disregarding the mechanism and cost of communication between the processors and the external memory. This facilitates the design and analysis of parallel algorithms.

The standard *ideal* (CRCW) PRAM has the following properties:

1. The processors are tightly synchronized, with no explicit cost of synchronization;
2. Every processor is always operational;
3. Every memory cell can be accessed by any processor;
4. Every memory cell can be accessed in one step;
5. Every memory cell can be read from or written to with no errors occurring.

* This research was partially supported by EC Cooperative Action IC-1000 (project ALTEC: *Algorithms for Future Technologies*).

Recently there has been a lot of research done concerning PRAM variants obtained by dropping or relaxing some of these properties. Asynchronous PRAMs were investigated in [6, 9, 17, 20]. PRAMs with faulty processors were studied by Kanellakis and Shvartsman [15] and Kedem *et al.* [16]. PRAMs with a differentiated cost of access to memory were considered by Aggarwal *et al.* [2] and Gibbons [9]. Valiant [24] considered the XPRAM model, where processors have a direct access only to their local memory, and access other cells by passing messages to the respective processors. There is also a closely related model of *distributed memory machine*, see a survey paper by Meyer auf der Heide [8].

Issues of distributed computing with faulty shared memory have been investigated by Afek *et al* [1] and Jayanti *et al* [13]. The problem of exploring the use of randomization to tolerate memory failures in synchronous models was posed by Afek *et al.* [1], and in asynchronous models with large granularity by Aumann *et al.* [3].

In this paper we consider a *faulty-memory* PRAM. Except for possible memory faults, this model has all the remaining properties of the ideal PRAM: it is fully synchronized, each shared memory cell can be accessed in one step by any processor, and the processors are always operational. A read instruction places the read value in a designated register of the processor, and similarly, a write instruction places the value stored in a register into the accessed memory word, unless it is faulty. There is a mechanism to react to errors in read and write operations, as follows. If the accessed memory word is faulty, then some designated register shows a special "faulty" value, and the respective processor knows that the attempted instruction failed due to a memory error.

We present simulation algorithms, which emulate the ideal PRAM on the faulty-memory machine. They are Monte-Carlo algorithms, that is, they are randomized and correct with a high probability. More precisely, they always operate within the stated time bounds, but may produce incorrect results with a small probability.

The rest of the paper is organized as follows. In Section 2, we introduce notations and concepts, and discuss the models of faults occurrence. The operations of broadcasting and spreading are described in Section 3. Static faults are handled by the algorithms A and B presented in Section 4, and dynamic faults by the algorithms C and D of Section 5. Conclusions and further research are discussed in Section 6.

Proofs of the theorems will be described in the final version.

2 Preliminaries

The PRAM model considered in this paper is the Concurrent-Read Concurrent-Write (CRCW) one. There are two specific variants used. In Collision, if many (more than one) processors attempt to write to a memory cell then a special collision symbol gets written. In Collision$^+$, if many processors attempt to write to a memory cell, then there are two cases: if all the values of the processors are equal then this common value gets written, otherewise the collision symbol is written to the cell. The algorithms of section 3 and 4 are designed for Collision, and of section 4 for Collision$^+$. See [5] for more on the relative power of these variants of the CRCW PRAM.

We use the following notations. The simulated ideal PRAM is denoted by C_I, and the simulating faulty-memory PRAM by C_F. Two main parameters of a simulation algorithm are the size of memory and the number of processors. The machine C_I has

n processors: p_1, p_2, ..., p_n, and n memory cells: s_1, ..., s_n. The machine C_F has N processors: P_1, P_2, ..., P_N, and M memory cells: S_0, ..., S_{M-1}. The number M is always assumed to be greater than n. A PRAM cell stores $O(\log n)$ bits. A processor of a PRAM has its own local memory, its cells are referred to as *registers*. The machine C_I has $O(1)$ registers per processor. All the registers of processors of C_F are assumed to be always operational and fully reliable. A processor of C_F has also $O(1)$ registers. Memory words of C_F are sometimes *marked*. This means setting some bits of them to specific values, the remaining bits to be used to simulate memory words of C_I.

There are two criteria by which (shared) memory errors are categorized.

Static versus Dynamic: if certain memory cells have become faulty before a computation starts, and no new faulty cells occur during the course of a computation, then the errors are *static*, otherwise the faults are called *dynamic*.

Deterministic versus Probabilistic: In the probabilistic case, each memory cell can be faulty with some fixed probability q, and any two cells are faulty independently of each other. In the deterministic version, at most $q \cdot M$ memory cells of C_F are faulty, for a constant parameter q, where $0 < q < 1$.

These two classifications are independent of each other, and this creates four settings for errors: deterministic static, deterministic dynamic, probabilistic static, and probabilistic dynamic. These combinations require further explanations. Conceptually, a setting can be imagined to be created in two steps. First, it is decided which cells are (potentially) faulty: in the deterministic case by selecting a subset of $q \cdot M$ elements from among the memory cells of C_F, and in the probabilistic setting, by deciding randomly and independently for each cell, whether it can be faulty. In the static case, all the selected cells become faulty, and in the dynamic case, it is assumed that there is an adversary who selects the step of simulation at which a given cell becomes faulty.

All the algorithms developed in this paper use randomization, and they can handle deterministic errors. They are automatically good against probabilistic memory failures, and hence probabilistic errors are not discussed as a special case. In what follows, it is always assumed that the faults are *deterministic*.

Similar models of error occurrence have been considered in the literature in the case of faulty processors. Probabilistic and dynamic faults of processors were studied by Kedem *et al.* [16] and Diks and Pelc [7]. Martel *et al.* [17] designed randomized simulations for deterministic processor errors.

For every presented simulation, the simulating machine C_F has to have the size of its memory within certain bounds for the simulation to run in the specified time with a large probability. To express this concisely, the notation $g(n) = \Theta^*(f(n))$ is used. It means that the inequalities $c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)$ hold, for a *sufficiently large* constant c_1 and $c_2 > c_1$. The criterion for being "sufficiently large" depends on the context. For instance, if C_F is said to have the memory of size $M = \Theta^*(f(n))$, then "sufficiently large c_1" may be a function of both the constant q, determining the number of faulty memory cells of C_F, and the required bound $1 - n^{-\alpha}$ on the probability.

Simulations are divided into two phases: formatting and the simulation proper. During formatting the memory is explored and organized, typically to construct a mechanism to access (some of) the operational memory cells. Then follows the proper part of simulation, performed in a step-by-step fashion. A step of C_I consists of three

parts: reading a memory cell, internal computation, writing to the same memory cell, and this is mimicked by C_F. The time of formatting is denoted by T_f, and the time of a one-step simulation by T_s, which is also referred to as the *step overhead*. The simulation algorithms are Monte Carlo and no synchronization between consecutive steps of C_I is needed. So algorithms operating in time t on C_I can be simulated on C_F to run in time $T_f + T_s \cdot t$, and be correct with a high probability.

A property $\Phi(n)$ is said to hold *with a high probability* (abbreviated to whp) if, for some $\alpha > 0$, $\Phi(n)$ holds with the probability at least $1 - n^{-\alpha}$, for a sufficiently large n. Throughout the paper, whenever this phrase is used, or the expression $1 - n^{-\alpha}$ is used explicitly as a bound on the probability, the value of α can be made arbitrarily large by manipulating other constants, for instance those involved either in the bounds on the number of memory cells, or the formatting time, or the step overhead.

3 Broadcasting and Spreading

In this section we consider two tasks that will be often performed during simulations. *Broadcasting* propagates a value known by one processor to the other processors. *Spreading* distributes $\log n$ values, known by $\log n$ processors, through the memory such that every value occupies $\Omega(M/\log n)$ cells.

ALGORITHM BROADCAST-1

Each processor repeats the following two steps $c_1 \cdot \log n$ times, for a constant c_1.

1. If the value v is not known then select randomly a memory cell S and read S. If S contains v then learn v.

2. If v is known then select randomly a memory cell S and write v to S.

Theorem 1. *If $N = \Theta(n)$ and $M = \Theta(n)$ then algorithm* BROADCAST-1 *operates in time $O(\log n)$ to propagate the value v among all the processors with the probability $1 - n^{-\alpha}$, for $\alpha > 0$.* □

The broadcasting algorithm may be also used when $M = O(n)$ does not hold. In this case each processor performs the body of algorithm BROADCAST-1, that is, its two steps, $O(\left(\sqrt{M/N} + 1\right) \cdot \log N)$ times. This algorithm is called BROADCAST-2.

Theorem 2. *Algorithm* BROADCAST-2 *propagates successfully the given value among all the processors, with the probability $1 - n^{-\alpha}$, for $\alpha > 0$.* □

Suppose that processor P_i knows value v_i, for $1 \leq i \leq \log n$, where $v_i \neq v_j$ for $i \neq j$. Then spreading is accomplished by the following algorithm:

ALGORITHM SPREAD

1. For each $1 \leq i \leq \log n$, processor P_i writes v_i into $c_2 \log n$ randomly chosen memory cells, for a constant c_2.

2. For each $1 \leq i \leq N$, processor P_i repeats $c_3 \log n$ times, for a constant c_3:

   2.1 If no value is known then select randomly a memory cell S and read it; if S contains v_j then learn v_j; otherwise

2.2 If some value v_j is known then select randomly a memory cell S and write v_j to S.

Theorem 3. *If $N = \Theta(n)$ and $M = \Theta(n)$ then, after $O(\log n)$ steps of algorithm* SPREAD, *every v_i occurs in $\Omega(n/\log n)$ operational cells with the probability $1 - n^{-\alpha}$, for $\alpha > 0$.* □

4 Static Faults

In this section we consider static faults: once a memory cell is checked to be non-faulty, it is guaranteed to remain such through the whole computation. The first algorithm is used later as a subroutine. It builds a binary tree over n cells. Once this is done, the address of the root is made known to every processor, and accessing the ith cell is performed by traversing the tree to the ith leaf.

ALGORITHM T

1. Each processor P_i selects randomly an operational memory cell x_i: this is done by repeatedly reading random memory cells, until such an operational one is found that is not claimed by other processors. Then P_i marks x_i.

2. Processor P_1 broadcasts the address of x_1 to all the remaining processors.

3. All the processors build a binary tree: First, cell x_1 is used to store the addresses of x_2 and x_3. Then, iteratively, the cell x_i is used to store the addresses of x_{2i} and x_{2i+1}.

Lemma 4. ALGORITHM T *can be implemented to run successfully in time $O(\log n)$ with the probability $1 - n^{-\alpha}$, for $\alpha > 0$.* □

4.1 Algorithm A

Suppose that the simulating machine C_F has $N = n \log n$ processors and $M = \Theta^*(n)$ memory cells. The presented simulation has $O(1)$-time step overhead with a high probability.

Divide the processors of C_F into n groups: $\mathcal{P}_1$, $\mathcal{P}_2$, ..., $\mathcal{P}_n$, each consisting of $\log n$ processors. Group $\mathcal{P}_i$ is to simulate the ith processor p_i of C_I. Let U_l denote the memory cell of C_F simulating the lth memory cell s_l of C_I. To locate U_l, the processors in a group compute addresses $D_1(l)$, $D_2(l)$, ..., where the *address function* D_j is defined as $D_j(x) = x + d_j \pmod{M}$, for the numbers d_j being random elements from the interval $[0, M-1]$. Only one of these addresses is designated as U_l, even if many are of operational memory cells. To visualize the underlying idea, consider the bipartite graph with edges (x, y), where x is the address of a memory cell in C_I and y is of the form $D_j(x)$, and the memory cell with address $D_j(x)$ in C_F is operational. Then a perfect matching in this graph gives a viable addressing scheme on C_F to simulate C_I. For every group $\mathcal{P}_i$ there is a special memory cell used for inter-processor communication, denoted by c_i. There are $d \log n$ address functions, where the number d is a parameter. The following algorithm initializes the memory such that among the memory words $D_j(l)$, for a fixed $1 \le j \le d \log n$, exactly one (operational) is marked as U_l.

ALGORITHM A (FORMATTING)

1. Processor P_1 generates $d \log n$ random numbers $d_1, d_2, \ldots, d_{d \log n}$ and places them in the shared memory organized as a list. This is performed by picking memory cells at random to place consecutive elements of the list.

2. The address of the header of the list is broadcast to all the processors by the algorithm BROADCAST-1.

3. The processors scan the list, and each processor, which is kth in his group, remembers the kth block of d values in the list.

4. A binary tree with n leaves is built by executing algorithm T. The ith leaf is the communication cell c_i for group $\mathcal{P}_i$.

5. For each $0 \leq k \leq m$, the kth cell U_k is selected from among the addresses $D_1(k)$, $D_2(k)$, ..., $D_{d \log n}(k)$, and marked as such. This is done as follows. First the addresses $D_1(1)$, $D_1(2)$, ..., $D_1(n)$ are examined, then $D_2(1)$, $D_2(2)$, ..., $D_2(n)$, and so on. For each l, the first $D_j(l)$, which is operational and not reserved already gets marked as U_l and then written to c_l to notify the remaining processors in $\mathcal{P}_l$.

In the step simulation, the task is to have the processors in some group $\mathcal{P}_i$ access a memory cell U_l.

ALGORITHM A (STEP SIMULATION)

1. Each processor in a group attempts to read the cell with address $D_j(l)$, for each j that it learned in Step 3 of formatting.

2. The processor that accessed the cell designated as U_l writes its address to c_i.

3. The remaining processors in $\mathcal{P}_i$ read c_i and then access U_l.

Theorem 5. *Algorithm A can be implemented in such a way that the time of formatting is $T_f = O(\log n)$ with the probability $1 - n^{-\alpha}$, and, once the formatting is successful, the step overhead is $T_s = O(1)$.* □

4.2 Algorithm B

Let the simulating PRAM C_F have $N = n/\log n$ processors and $M = \Theta^*(n)$ shared-memory cells. The algorithm B has $O(\log n)$-time step overhead, which is optimal for the available number of processors.

Suppose that $M = c_M \cdot n$, for a constant $c_M > 0$. Divide $c_M \cdot n$ memory cells into contiguous blocks of $\beta \cdot k$ cells, where $k = \log n + \sqrt{\log n} + 1$ and $\beta > 1$. A block containing at least k operational cells is said to be *good*. The following inequality estimates the number l of such blocks:

$$l \geq c_M \cdot n \frac{k(\beta(1-q)-1)+1}{\beta \cdot k(k(\beta-1)+1)} \geq c_M \cdot n \frac{\beta(1-q)-1}{\beta(k(\beta-1)+1)}$$

Choose c_M and β in such a way that the inequality $l \geq n/\log n$ holds. The first $n/\log n$ good blocks B_0, B_1, ..., are used to simulate the memory of C_I. A block

B_i consists of the *root* $r(i)$, $\sqrt{\log n}$ *auxiliary cells*, and $\log n$ *normal cells*. Auxiliary cells are interspersed among normal cells and partition them into groups of $\sqrt{\log n}$ elements. The jth normal cell of B_i simulates the cell $i \cdot \log n + j$ of C_I. The bits of the root of a block are divided into $\sqrt{\log n}$ fields F_1, F_2, ..., a field F_j stores the offset of the jth auxiliary cell in the block. Similarly, an auxiliary cell has its bits divided into $\sqrt{\log n}$ fields to store offsets of the next $\sqrt{\log n}$ normal cells. The *offset* of an auxiliary cell x is the number of cells between the beginning of the block and x; and for a normal cell y, the *offset* of y is the number of cells between y and the auxiliary memory cell closest to the left. Notice that an offset is a number of size $O(\log n)$, hence it requires $\log\log n + O(1)$ bits to be stored. Since a memory word is assumed to be able to store $O(\log n)$ bits, all the offsets fit into the roots and auxiliary cells. Given the address of a root $r(i)$, the jth normal cell in B_i can be accessed in time $O(1)$ by first locating the respective auxiliary cell, and then the normal cell. To this end, the fields of root and the respective auxiliary cell are extracted by applying standard arithmetic and boolean bit operations. We show next how to make the roots accessible in time $O(1)$ with a high probability. The address functions D_1, D_2, ... are defined similarly as in algorithm A, the number d is a parameter.

ALGORITHM B (FORMATTING)

1. The blocks are divided into $N = n/\log n$ groups. Processor P_i counts the number of good blocks in the ith group.

2. A binary tree of $M/\beta \log n$ leaves is built by executing algorithm T. The tree is used next to assign consecutive numbers to good blocks as in the parallel prefix algorithm.

3. Processor P_i sets the root and the auxiliary cells in block B_i to their proper values.

4. Processor P_1 generates a list of $d \log n$ random numbers $d_1, \ldots, d_{d \log n}$ and organizes them as a list, similarly as in algorithm A.

5. Processor P_i evaluates the address functions $D_1(i), \ldots, D_{d \log n}(i)$. For every l, if the cell $D_l(i)$ is operational and unmarked, then P_i marks it and sets to store a pointer to the root $r(i)$.

6. Each processor P_i creates a list of $\log n$ memory cells to store the contents of registers of the processors that it is to simulate.

ALGORITHM B (STEP SIMULATION)

Each processor P_l simulates $\log n$ processors of C_I. The contents of registers are stored in the list that P_l built in the last step of formatting. P_l scans the list and retrieves the addresses x_1, x_2, ... of memory cells that need to be accessed. To locate x_1, processor P_l tries addresses $D_1(i_1)$, $D_2(i_1)$, ..., $D_{t_1}(i_1)$, where i_1 is the number of block containing x_1. The search terminates when a memory cell with a pointer to the root is found. Then P_l is able to access x_1 in time $O(1)$. Processor P_l continues with $D_{t_1+1}(i_2)$, $D_{t_1+2}(i_2)$, ..., $D_{t_2}(i_2)$, terminating when a pointer to the second root is found. The remaining addresses are processed similarly, each time starting with the first unused yet address function.

Theorem 6. *Algorithm B can be implemented in such a way that the time of formatting is equal to* $T_f = O(\log^{3/2} n)$ *with the probability* $1 - n^{-\alpha}$*, and the step overhead is* $T_s = O(\log n)$*, with the probability* $1 - n^{-\alpha}$ *for each step, for* $\alpha > 0$. □

5 Dynamic Faults

Dynamic faults are more challenging. In such a setting some form of duplication or dispersal of information stored in the memory of C_F is inevitable, since a memory cell storing useful information may turn out to be faulty at any time. To simplify the presentation and analysis, we assume that a memory word of C_F may store the contents of several words of C_I.

5.1 Algorithm C

Let the simulating PRAM have $N = n \log^2 n$ processors and $M = \Theta^*(n \log n)$ memory cells. We present a simulation that has a $O(1)$-time step overhead.

The number d is a parameter. To simplify the notation, assume rather that there are exactly $N = d^2 \cdot n \cdot \log^2 n$ processors of C_F. Divide these processors into n groups $\mathcal{P}_1$, ..., $\mathcal{P}_n$ of $d^2 \cdot \log^2 n$ processors. Divide each group $\mathcal{P}_i$ into subgroups $\mathcal{P}_{i,j}$ of $d \cdot \log n$ elements. The task of $\mathcal{P}_i$ is to simulate the processor p_i of C_I. It is assumed that, when the simulation starts, all the processors from group $\mathcal{P}_i$ know the initial state of processor p_i. In the course of the simulation, typically only a fraction of processors in every group $\mathcal{P}_i$ know the current state of p_i, these processors are called *informed*. Each informed processor knows that it is informed, otherwise it is aware that it is not. After the computation of C_I terminates, it may still take some time for all the processors of C_F to get informed. This time period is called the *termination delay*.

ALGORITHM C (FORMATTING)

1. The processors generate $d \cdot \log n$ random numbers d_i in $[0, M-1]$.

2. The numbers d_i are distributed through the shared memory by executing the algorithm SPREAD on $O(n \log n)$ processors.

3. Each processor P_k makes $O(\log n)$ attempts to learn some of the numbers d_i by reading memory cells selected at random. It stores the first $r = O(1)$ of them, denote the respective address functions by D_1^k, ..., D_r^k. The number r is called a *fan*, and is a parameter of the algorithm. If the processor is in $\mathcal{P}_{i,j}$ and it happens to get to know d_j, then it stores d_j.

4. Steps 1 through 3 are repeated, for a new set of $d \cdot \log n$ numbers denoted g_j. Processor P_k learns the functions G_1^k, ..., G_r^k.

In Step 2, notice that whp there are only $O(1)$ pairs $i \neq j$ such that $d_i = d_j$, so Theorem 3 still holds. The generated random numbers define address functions $D_i(x) = x + d_i \,(\mathrm{mod}\, M)$, and $G_i(x) = x + g_i \,(\mathrm{mod}\, M)$. A memory cell $D_j(x)$ ($G_j(x)$, resp.) which is operational and such that its number is the value of only one of the functions D or G for exactly one argument is said to be *D-useful* (*G-useful*, resp.). D-useful cells simulate cells of C_I , G-useful cells are used by groups of processors for

communication. We can mark each written value with the number of the simulated cell, in the case of D-usefulness, or of the processor group, in the case of G-usefulness. Then a cell can be verified as not being D-useful or G-useful if it is either not operational or contains the collision symbol or does not contain the correct number.

Suppose that P_i needs to access s_l in the current step.

ALGORITHM C (STEP SIMULATION)

1. (Read) If a processor P_k in $\mathcal{P}_i$ is informed, that is, P_k knows l, then it attempts to read the memory cells $D_1^k(l)$, ..., $D_r^k(l)$.

2. If a processor P_k in $\mathcal{P}_{i,j}$ succeeded in Step 1 (that is, read at least one D-useful cell), and P_k knows g_j, then P_k writes the contents of its registers to $G_j(i)$.

3. If a processor P_k in $\mathcal{P}_{i,j}$ failed to read from an operational memory cell in Step 1 (during this iteration, or in one of the previous iterations and has not succeeded in Step 3 since then) then it attempts to read (at least one of) $G_1^k(i)$, ..., $G_r^k(i)$.

4. If a processor P_k in $\mathcal{P}_i$ failed in Steps 1 and 3, then it selects a memory cell C at random and attempts to read it. If C stores a number g_a, then P_k attempts to read $G_a(i)$. If the state of computation stored there means termination, then P_k stops.

5. (Write) If a processor P_k in $\mathcal{P}_{i,j}$ succeeded in either Step 1 or Step 3, then it first performs the internal computation of C_I, and next writes to $D_j(l)$.

Denote $D(x) = \{D_i(x) : 1 \leq i \leq d \log n\}$, for $0 \leq x \leq n-1$, and define $G(x)$ similarly.

Lemma 7. *There are at least* $\frac{(1-q)}{2} \cdot d \log n + 1$ *cells in* $D(x)$ *that are D-useful, with the probability* $1 - n^{-\alpha}$, *for* $\alpha > 0$, *for a sufficiently large* $M = \Theta^*(n \log n)$. *The same fact holds for* $G(x)$. □

Let $G = (A, B, E)$ be a bipartite graph, where E is the set of edges connecting elements of A and B. Graph G is said to have the (γ, β) *weak-expansion property*, for $0 < \gamma, \beta < 1$, if, for every set $X \subseteq A$ such that $|X| \geq \gamma|A|$, the set $\Gamma(X) = \{y \in B : (x, y) \in E\}$ satisfies $|\Gamma(X)| \geq \beta|B|$. We will consider specific bipartite graphs defined as follows: $A = \mathcal{P}_{i,j}$, $B = D(x)$, there is an edge connecting a processor P in A with $D_a(x)$ iff D_a is among the address functions known by P after formatting. This graph is denoted $\mathcal{D}_{x,i,j}$. A similar graph $\mathcal{G}_{i,j}$ is defined as follows: $A = \mathcal{P}_{i,j}$, $B = G(i)$, there is an edge connecting a processor P in A with $G_a(i)$ iff G_a is among the address functions known by P after formatting.

Lemma 8. *Graphs* $\mathcal{D}_{x,i,j}$ *and* $\mathcal{G}_{i,j}$ *have the* (γ, β) *weak-expansion property, for any* $0 < \gamma, \beta < 1$, *with the probability at least* $1 - n^{-\alpha}$, *for* $\alpha > 0$ *and a sufficiently large fan* r. □

The performance of algorithm C is estimated in the following theorem:

Theorem 9. *Algorithm C has the formatting time* $T_f = O(\log n)$, *the step overhead* $T_s = O(1)$, *and the termination delay* $O(\log n)$, *all with the probability at least* $1 - n^{-\alpha}$, *for* $\alpha > 0$. □

5.2 Algorithm D

Suppose that C_F has $N = n$ processors and $M = \Theta^*(n)$ memory cells. To be specific, $M = c_M \cdot n$. It is assumed that all the processors of C_F know a primitive element of $GF(u)$, where $u > \log n / \log\log n$ is a power of 2. During a simulation, the contents of a memory cell of C_I are encoded, divided into pieces, and then distributed among $d \log n$ memory cells of C_F, for a constant parameter d. The numbers a and c are also parameters.

ALGORITHM D (FORMATTING)

1. Processors P_i generate $d \log n$ random numbers d_i in $[0, M-1]$, and $a \log n$ random numbers a_j in $[0, M-1]$.

2. The numbers a_j are distributed in the memory by executing SPREAD.

3. Each processor P_i, for $1 \leq i \leq d \log n$, repeats $c \log n$ times of the following two steps: select randomly memory cell and read it; if a_j was read then write d_i to the cell number $(a_j + i) \mod M$.

We recall some facts from the theory of error correcting codes, consult [4, 19] for more information. Let C be sequence of codes $C = C_1 \ldots C_n \ldots$, where $C_n \subseteq \Sigma^n$, for some alphabet Σ of size s. Let $C_n(j)$ denote the jth codeword of code C_n. C is called *asymptotically good* if the lengths n, sizes $M_n = |C_n|$, dimensions $m_n = \lfloor \log_s M_n \rfloor$ and minimum Hamming distances d_n of codewords from C_n satisfy the following: the *rate* of the sequence $R = \liminf_{n\to\infty} \frac{m_n}{n}$, and the *relative minimum distance* $\delta = \liminf_{n\to\infty} \frac{d_n}{n}$ are both strictly greater then zero. By the Gilbert-Varshamov bound (see [4, 19]), for any $\delta \in [0, 1 - \frac{1}{s})$, there exists an asymptotically good code such that $R > 0$, that is, $R \geq 1 - H_s(\delta)$, where $H_s(x) = -x \log_s x - (1-x) \log_s (1-x) + x \log_s (s-1)$. It is also known, see [19], that for $n = s - 1$ there exist codes, called Reed-Solomon (or simply RS) codes, with $\delta \geq 1 - R$. These codes can be quickly encoded and decoded, and are used in our algorithm. We apply a technique called "concatenation" of codes, in which the final code is obtained by first encoding by a code with a large alphabet size (outer code), and then encoding each symbol using another code (inner code). Notice that if a code C_n has the minimum distance d_n then it can correct n_ϵ erasures (eliminations of symbols) and n_e errors (changes of symbols) provided that $2n_e + n_\epsilon < d_n$. Suppose that processor P needs to simulate the instruction of writing some value w of $\log n$ bits into s_l. The following is a high level description of the algorithm. Coding and decoding need to be performed as bit operations on words. The details will be given in the final version.

ALGORITHM D (STEP SIMULATION, WRITE)

1. Divide the word w into blocks w_1, w_2, ... of $\log u$ consecutive bits and encode it as v_1,v_2,... by the RS code over $GF(u)$ with a relative minimum distance β, for a suitable β.

2. Encode each v_i by a suitable asymptotically good code C.

3. Let $k = \lfloor l / \log n \rfloor$. Attempt to store the consecutive symbols of the word $C(v_1), C(v_2) \ldots$ in $D_1(k)$, $D_2(k)$, ..., on the position $l \mod \log n$. In every trial select $r = O(1)$

algorithm	errors	#processors	memory size	formatting time	step overhead
A	static	$n \log n$	$\Theta^*(n)$	$O(\log n)$	$O(1)$
B	static	$n/\log n$	$\Theta^*(n)$	$O(\log^{3/2} n)$	$O(\log n)$
C	dynamic	$n \log^2 n$	$\Theta^*(n \log n)$	$O(\log n)$	$O(1)$
D	dynamic	n	$\Theta^*(n)$	$O(\log n)$	$O(\log n)$

Table 1. A comparison of the four simulations developed, in terms of their resources available and time performance. For the definition and explanation of notation Θ^* see section 2.

random cells. If a value a_j was found, try to read d_i from a cell number $(a_j + i)$ mod M.

ALGORITHM D (STEP SIMULATION, READ)

1. As in Step 3 of the write part, read the cells $D_1(k), D_2(k) \ldots$ and form a sequence of codewords $z_1, z_2 \ldots$ (possibly with errors and erasures).
2. Decode every z_i.
3. Apply the RS decoding algorithm to the sequence obtained in Step 2.

The performance of algorithm D is estimated by the following theorem:

Theorem 10. *Algorithm D can be implemented in such a way that the formatting time T_f and the step overhead T_s are both $O(\log n)$, with the probability $1 - n^{-\alpha}$, for $\alpha > 0$.* □

6 Remarks

Four simulations of an ideal fully-reliable PRAM on a faulty-memory PRAM have been developed. Two settings of error occurrence are considered: static and dynamic. Given the kind of errors, there are two parameters of simulations: the number of processors, and the size of memory of the simulating PRAM. The performance of a simulation is measured by the formatting time and the step overhead. All this information is collected in Table 1. The simulation B is close to optimal, in the sense that the work done after formatting is $O(n \cdot t)$, where t is the time of the simulated algorithm on C_I. In general, the optimality of the presented algorithms, for the given resources, is an open problem. One could set some of the performance measures as targets, and try to minimize the other ones. Our choice was to design $O(1)$-time and $O(\log n)$-time step-overhead simulations while minimizing the formatting time, the number of processors and the size of the shared memory. It seems that the time cost of any simulation must be at least logarithmic, and proving such a lower bound would be interesting. The data compiled in Table 1 are consistent with this hypothesis, since the two rightmost columns contain at least one logarithm in every row.

There is an alternative method to algorithm D in which the information dispersal of Rabin [21] is used instead of the RS codes and asymptotically good codes. It requires $\log n$ registers per processor to have a time performance comparable to algorithm D.

All the algorithms described in this paper can be adapted to a situation when the simulated PRAM has $m > n$ memory cells, where n is the number of processors. Then the time of a simulation of one step remains the same, and the formatting time is multiplied by at most $O(m/n)$. By checking the correctness of computations after each simulated step, the presented algorithms may be converted to be Las Vegas. Such checking may be performed by counting all the processors that performed the simulation correctly. Details will be presented in the full version of this paper.

This research is in the line of studying the PRAM model with weaker properties than the classical ideal version, for instance by allowing faults in hardware. We concentrated on the CRCW PRAM. It would be interesting to study weaker models, like CREW or EREW, and also the case when both the processors and memory cells may be faulty.

References

1. Y. Afek, D.S. Greenberg, M. Merrit, and G.Taubenfeld, Computing with Faulty Shared Memory, Proc. 11th Ann. Symposium on Principles of Distributed Computing (1992), 47-58.
2. A. Aggarwal, A.K. Chandra, and M. Snir, On Communication Latency in PRAM Computations, Proc. 1st Ann. ACM Symposium on Parallel Algorithms and Architectures (1989), 11-21.
3. Y. Aumann, Z.M. Kedem, K.V. Palem, and M.O. Rabin, Highly Efficient Asynchronous Execution of Large-Grained Parallel Programs, Proc. 34th Ann. Symposium on Foundations of Computer Science (1993), 271-280.
4. N. Alon, J. Bruck, J. Naor, M. Naor, and R.M. Roth, Construction of Asymptotically Good Low-Rate Error-Correcting Codes through Pseudo-Random Graphs, IEEE Trans. Inf. Theory, 38 (1992), 509-516.
5. B.S. Chlebus, K. Diks, T. Hagerup, and T. Radzik, New Simulations between CRCW PRAMs, Proc. 7th International Conference on Fundamentals of Computation Theory (1989), 95-104, Springer LNCS 380.
6. R. Cole, and O. Zajicek, The APRAM: Incorporating Asynchrony into the PRAM Model, Proc. 2nd Ann. ACM Symposium on Parallel Algorithms and Architectures (1990), 158-168.
7. K. Diks, and A. Pelc, Reliable Computations on Faulty EREW PRAM, manuscript, 1993.
8. F. Meyer auf der Heide, Hashing Strategies for Simulating Shared Memory on Distributed Memory Machines, Proc. of the 1st Heinz Nixdorf Symposium "Parallel Architectures and their Efficient Use,"(1992), 20-29, Springer LNCS 678.
9. P.B. Gibbons, A More Practical PRAM Model, Proc. 2nd Ann. ACM Symposium on Parallel Algorithms and Architectures (1990), 169-178.
10. A.M. Gibbons and W. Rytter, "Efficient Parallel Algorithms," Cambridge University Press, 1988.
11. T. Hagerup and Ch. Rüb, A Guided Tour of Chernoff Bounds, Inf. Proc. Letters 33 (1989/90), 305-308.
12. JáJá, "An Introduction to Parallel Algorithms," Addison-Wesley, 1992.
13. P. Jayanti, T.D. Chandra, and S. Toueg, Fault-tolerant Wait-free Shared Objects, Proc. 33rd Ann. Symposium on Foundations of Computer Science (1992), 157-166.
14. J. Justesen, On the Complexity of Decoding Reed-Solomon Codes, IEEE Trans. Inf. Theory, 22 (1976), 237-238.
15. P.C. Kanellakis and A.A. Shvartsman, Efficient Parallel Algorithms Can Be Made Robust, Distributed Computing, 5 (1992), 201-217.

16. Z.M. Kedem, K.V. Palem, and P.G. Spirakis, Efficient Robust Parallel Computations, Proc. 22nd ACM Symp. on Theory of Computing (1990), 138-148.

17. Ch. Martel, R. Subramonian, and A. Park, Asynchronous PRAMs Are (Almost) as Good as Synchronous PRAMs, Proc. 31st Ann. Symposium on Foundations of Computer Science (1990), 590-599.

18. Mc Diarmid, On the Method of Bounded Differences, in J. Siemon, ed., "Surveys in Combinatorics,", 148 - 188, Cambridge University Press, 1989, London Math. Soc. Lecture Note Series 141.

19. F.J. MacWilliams, and N.J.A Sloane, "The Theory of Error-Correcting Codes," North-Holland, 1977.

20. N. Nishimura, Asynchronous Shared Memory Parallel Computation, Proc. 1st Ann. ACM Symposium on Parallel Algorithms and Architectures (1989), 76-84.

21. M.O. Rabin, Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance, Journal of ACM, 36 (1989), 335-348.

22. D.V. Sarwate, On the Complexity of Decoding Goppa Codes, IEEE Trans. Inf. Theory, 23 (1976), 515-516.

23. Y. Sugiyama, M. Kosahara, S. Hirasawa, and T. Namekawa, An Erasures and Error Decoding Algorithm for Goppa Codes, IEEE Trans. Inf. Theory, 22 (1976), 238-241.

24. L.G. Valiant, General Purpose Parallel Architectures, in J. van Leeuwen, ed., "Handbook of Theoretical Computer Science," vol. A, 943-971, Elsevier, 1990.

This article was processed using the LaTeX macro package with LLNCS style