Fast Deterministic Computations on a Faulty PRAM

Bogdan S. Chlebus [*] Leszek Gąsieniec [*†] Andrzej Pelc [†]

Abstract

We develop a method to simulate deterministically the operational Parallel Random Access Machine (PRAM) on a PRAM prone to processor and memory failures. It is shown that an n-processor PRAM with a bounded fraction of faulty processors and memory cells can simulate deterministically the fully operational PRAM with $\mathcal{O}(\log n)$ slowdown, after preprocessing performed in time $\mathcal{O}(\log^2 n)$.

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

1 Introduction

The increase of the number of processing elements in multiprocessor computers makes the machines prone to hardware failures. This poses the task for programmers to make algorithms resilient to faults encountered in the course of a computation. The most general solution is to make any algorithm fault-resilient by providing a simulation mechanism.

The model of computation that we consider in this paper is the *Parallel Random-Access Machine* (PRAM) (see [4, 5, 8]). It consists of a number of synchronized processors and a global shared memory. Each processor has its own local control and its own local memory. The global memory is assumed to consist of individually addressable memory cells. Each processor has the complete capabilities of a RAM, and it can access arbitrary locations both in the global shared memory and its local memory in a single step.

There is a family of PRAM models classified according to the ability of processors to access the shared memory.

- The weakest *exclusive-read exclusive-write* EREW allows only at most one processor to read or write to any location in the shared memory during each step.
- The *concurrent-read exclusive-write* CREW allows many processors to read from the same location in the shared memory at the same time, but simultaneous writes are not allowed.
- The strongest *concurrent-read concurrent-write* CRCW allows for simultaneous reads and writes by many processors to the same locations in the shared memory. Now some method of arbitrating write conflicts needs to be specified.
  - The weakest COMMON CRCW allows concurrent writes only when all the processors are attempting to write the same value to the memory location.
  - The stronger ARBITRARY CRCW allows an arbitrary processor to succeed during simultaneous write.
  - The strongest PRIORITY CRCW operates with some underlying linear ordering of all the processors. During a simultaneous write, the smallest (according to the given order) processor succeeds.

For the CREW and EREW PRAMs, violating the exclusivity restrictions means a runtime error. Likewise, for the COMMON CRCW, concurrent write attempts of different values to the same memory cell result in a runtime error.

There has been a lot of research done recently to develop simulation techniques transforming PRAM algorithms designed to operate in a fault-free environment into

algorithms that are reliable on fault-prone machines. Many approaches have been applied, depending on the nature of faults (static versus dynamic, deterministic versus stochastic, fail-stop versus restartable), the efficiency criteria (time versus work), the properties and capabilities of the underlying model (synchronous versus asynchronous, concurrent read and write versus exclusive read and write), or the types of simulations (randomized versus deterministic, definite versus tentative). Kanellakis and Shvartsman [6] introduced the fail-stop PRAM and developed many deterministic and robust algorithms. This approach to fault tolerance was continued in a series of papers. Kedem, Palem and Spirakis [10] and Shvartsman [14] developed robust general PRAM simulations for dynamic fail-stop errors. Kedem *et al.* [9] produced robust randomized tentative simulations (without per-step synchronization) which have constant expected slowdown, under some assumptions on the stochastic faults. Kanellakis and Shvartsman [7] developed deterministic robust PRAM emulations under the model of restart failures. Chlebus, Gambin and Indyk [1] studied the CRCW PRAM with deterministic memory faults, and designed efficient randomized PRAM simulations for both static and dynamic errors. Diks and Pelc [3] developed reliable and efficient algorithms for the EREW PRAM, under the stochastic processor-fault model.

There is also a large literature on simulating PRAM on more realistic distributed-memory machines, see [11, 12, 15] and the references therein. We present a general deterministic method of simulating parallel algorithms in a faulty environment. It can be applied in any synchronous model of computation in which any two processors can communicate in constant time. In this paper the exposition is restricted to the PRAM model.

The developed simulation is of the fully operational PRAM on a PRAM with faulty memory and faulty processors. All the faults are deterministic and static. Here *deterministic* means that the simulation performance is with respect to any distribution of faults, and *static* means that the *faulty/operational* status of error-prone components does not change in the course of a computation. It is assumed that at most a bounded fraction of processors or memory cells are faulty. If an operational processor attempts to read a memory cell then it is immediately notified whether the operation was successful (fault-free addressee) or failed (faulty addressee).

The main contribution of this paper is showing that one can obtain a logarithmic slowdown in the worst case in simulations of the operational PRAM on a faulty one. More precisely, we prove the following main result:

Theorem 1 *An n-processor Parallel Random Access Machine with a bounded fraction of faulty processors and memory cells can simulate deterministically its fault-free counterpart in a step-by-step fashion with $\mathcal{O}(\log n)$ slowdown and preprocessing done in time $\mathcal{O}(\log^2 n)$.* □

The presented fault-free PRAM simulation on a PRAM with faulty shared memory

is related to the results obtained in [1]. There are two simulations developed in [1] for the static memory errors: one with a $\mathcal{O}(\log n)$ slowdown on the optimal number of $n/\log n$ processors, the other operating in real time on $n \cdot \log n$ processors. To compare the results notice that the simulations developed in [1] are randomized, the performance bounds are expected, and the model is a CRCW PRAM, whereas the simulation presented in this paper is deterministic and applicable to the whole PRAM family, in particular to the weakest EREW PRAM.

This is for the first time that a general time-efficient deterministic simulation technique is designed for the EREW PRAM with the worst-case fault distribution. At first glance, it could seem that *no* deterministic and fast simulation of a PRAM with faulty memory cells is possible, under the worst-case fault distribution. Namely, if the simulation is deterministic then the memory cells accessed by each processor are predetermined until the first operational cell is encountered. This implies that some operational processors may *never* (within the time bounds of a fast simulation) find a fault-free memory cell and hence may never be able to communicate with the other processors. To circumvent this obstacle, the developed simulation is designed in such a way that it relies only on a fraction of the operational processors. Now the question arises: what with the input supplied to processors that do not participate in the simulation? We solve this problem by adapting the method of information dispersal of Rabin [13] and show how all the original input can be retrieved from the information available to the processors being active in the simulation.

The paper is organized as follows. In Section 2 we describe the simulation problem and propose how to reduce it to two abstract tasks: construction of a list of fault-free processors and communication along this list. In Sections 3 and 4 we show how these two tasks can be performed on a faulty PRAM. In Section 5 we discuss the problem of how to supply the input to a faulty PRAM. Section 6 contains final remarks.

The results of this paper were reported in a preliminary form in [2].

2 Simulations on a faulty PRAM

We present simulations for all three variants of the PRAM model: EREW, CREW, and CRCW PRAMs. A simulation is in two phases: the preprocessing is followed by the simulation proper done in a step-by-step fashion. In this section we provide all the details and the setting of the problem of simulating the fault-free PRAM on its fault-prone counterpart. We propose to reduce the simulation to two abstract problems which are then discussed in detail in the following sections.

Two kinds of faults in the PRAM are considered: faulty processors and faulty cells in the global shared memory (the shared memory cells are simply called *cells* in what follows). Let n denote the number of processors, and let the processors be labeled 1

through n. For the sake of simplicity of exposition, let us assume that the number of cells in the simulated fault-free machine is also n. The fault-prone simulating machine has n processors and cn cells, for a constant $c > 1$ to be determined later. The consecutive segments of c cells each are assigned to consecutive processors. We assume that the fraction of faulty cells and the fraction of faulty processors are bounded from above by some constants q_1 and q_2 respectively, where $q = q_1 + q_2 < 1$. Let d be a sufficiently large constant. Call a processor *active* if it is fault-free and there are at least d fault-free memory cells in the segment of c cells assigned to it. Otherwise call it *dormant*. (An active processor needs a sufficiently large memory part of assigned memory to operate). We require that at least αn processors be active, for some constant $0 < \alpha \leq 1$, such that $d > \frac{1}{\alpha}$, because every active processor simulates the work of $\frac{1}{\alpha}$ processors.

Notice that there are at most $\frac{q_1 cn}{c-d}$ segments which contain less then d fault-free cells each. Another $q_2 n$ segments could be unreadable since their associated processors are subject to fail. Thus there are at least $\alpha n = n - \frac{q_1 cn}{c-d} - q_2 n$ segments corresponding to active processors. Since $\alpha = 1 - \frac{q_1 c}{c-d} - q_2$ and $\alpha > \frac{1}{d}$ we get

$$1 - \frac{q_1 c}{c-d} - q_2 > \frac{1}{d}$$

or equivalently

$$c > \frac{d(1 - q_2 - \frac{1}{d})}{1 - q_1 - q_2 - \frac{1}{d}}.$$

This means that for a sufficiently large constant c (not depending on n) an appropriate fraction αn of active processors can be guaranteed. Faults are *static*, in the sense that the *faulty/operational* status of the processors and the memory cells is the same during a computation. In particular, no new faults appear in the course of a computation. Each operational processor has a fully reliable local memory of size $O(1)$. A fault-free processor accessing a cell learns immediately about the fault status of the cell. This mechanism is an extension of the standard PRAM model; technically, we assume that there is a special one-bit register at each processor which is set to the value corresponding to the fault status of the accessed cell. Faulty processors do not perform any attempts of memory access.

During the preprocessing, a list of all the active processors is built. When the list has been constructed, every active processor knows the length l of the list, its own rank in it and the label of its successor. In Section 3 it is shown how this preprocessing can be done in time $O(log^2 n)$. For simplicity, let us assume that l divides n and let $m = n/l$. The ith active processor will simulate the actions of processors with labels in the set $S_i = \{(i-1)m+1, ..., im\}$. Suppose that, in a given computation step of the simulated algorithm, processor j_1 accesses the cell corresponding to processor j_2. Let

$j_1 \in S_{i_1}$ and $j_2 \in S_{i_2}$, $j_2 = (i_2 - 1)m + x$. In the faulty machine, the active processor A with rank i_1 in the list knows that it simulates the action of j_1 and can compute i_2 and x. The main component of the simulation of a step of computation consists in accessing the xth fault-free cell corresponding to processor B with rank i_2 in the list. The only information that A is missing is the physical identification (label) of processor B. Hence the simulation would be complete if every active processor could learn the label of another active processor knowing its rank in the list. We call this task *communication along the list*. In section 4 we prove that this task can be accomplished in logarithmic time, thus yielding a logarithmic slowdown of the simulated algorithm.

3 Building a list

There are n processors p_1, p_2, ..., p_n in the PRAM. It was argued before that the number of active processors is at least $\alpha \cdot n$, for a constant $0 < \alpha \leq 1$. The simulation is performed by the active processors. The dormant processors do not participate in the computations, except that when an active processor inquires a dormant processor about its status (by reading a cell in the appropriate memory segment), it will be immediately notified that it attempted to communicate with a dormant one.

A processor is said to be *active* if it is operational and there are at least d operational memory cells in the segment of c cells assigned to it.

A set S of processors is *good* if the number of active elements of S is at least $\alpha \cdot |S|$, where $|S|$ denotes the size of S. The *partition tree* (PT_n), for the given n processors, is defined as follows. It is a full binary tree with the nodes labeled by intervals of integers. The notation $[a..b]$ denotes the interval consisting of the integers x such that $a \leq x \leq b$. The root is labeled by $[1..n]$. If a node v is labeled by $[a..b]$ and v is an internal node then its left child is labeled by $[a..\lfloor (a+b-1)/2 \rfloor]$ and the right child is labeled by $[\lceil (a+b)/2 \rceil..b]$. The leaves are labeled by intervals $[a..b]$ of length $\lceil \frac{n}{2^h} \rceil$ or $\lfloor \frac{n}{2^h} \rfloor$, for $h = \lfloor \log(\alpha n) \rfloor$. A processor p_i is *associated* with node v if the index i belongs to the label of v. The processors associated with a node of the tree PT_n are referred to as the *group* of the node. A node is *good* if its group is good. Notice the following:

A) The root of PT_n is good;
B) If a node of PT_n is good then one of its children is good.

Based on this observation, the following key property of the partition trees follows:

Lemma 1 *There is a good leaf in PT_n such that the path from v to the root contains only good nodes.* □

We design an algorithm BUILD_LIST organizing all the active processors in a list. The algorithm works in phases corresponding to the levels of the tree PT_n. During a phase the processors are partitioned into groups associated with nodes at the corresponding level of the tree. A group is either *busy* or not. If a group is busy then:

A) Each processor in the group knows this;
B) The active processors in the group are organized in a list, and each element of the list knows its *rank*, that is, the distance from the head of the list;
C) Each element knows the number of elements in the list.

During the first phase of the algorithm the groups correspond to the leaves of PT_n. The processors in each group check each other and if the group is good then the processors arrange themselves in a list, compute ranks and the group becomes busy. In general, once a phase has been completed, the computation proceeds to the next one corresponding to the next level of PT_n towards the root. Consider such a phase. It consists of three parts. During the first part the busy groups corresponding to nodes being left children perform computations, while the other groups pause, in the second part the busy groups corresponding to right children perform computations, but only if their left-sibling group was idle during part one. Consider the first part. Each processor in a busy group G_1 is assigned a set of processors in the sibling group G_2 as follows. The processors in G_2 are partitioned into as many subsets as there are active elements in G_1, the sizes of subsets are constant (G_1 is good) and differing by at most one. The active processor with rank k in the list in G_1 is assigned the kth subset of G_2. Each active processor of G_1 checks on the assigned subset and includes all its active processors into the list. This completes part one. The second part is similar, but the roles between sibling groups are reversed. For each pair of sibling groups, if at least one of the groups is busy then after part two all the active processors in these groups are connected in a common list. This list is processed during the third part. The ranks and the size of the list are computed and broadcast to all the elements of the list by the standard procedure of pointer jumping. If the size of the list is large enough for the group of the parent node to be good then the group of the parent node becomes busy in the next phase.

In this description of the algorithm one detail is missing, namely how to control the duration of the first two parts of a phase, when the processors check on the assigned elements of sibling groups. Notice that this time can be bounded by a common constant since all the busy groups are good.

Theorem 2 *The algorithm* BUILD_LIST *produces a list connecting all the active processors, and operates in time* $\mathcal{O}(\log^2 n)$.

Proof. By Lemma 1, there is at least one busy group at each phase. The correctness follows by induction on the height of PT_n. There are $\mathcal{O}(\log n)$ phases. The first two parts of a phase take time $\mathcal{O}(1)$, but the third one takes time proportional to the logarithm of the size of the list, which is $\mathcal{O}(\log n)$. □

4 Communication along the list

In this section the word *processor* means *active* processor. Let us consider a specific step of computation. The processors are assumed to have already been connected in a list. The word *list* means the underlying directed list of processors. The indices of processors used in this section are their ranks in the list. Each processor $P(i)$ knows the index of another processor $P(j)$ that it wants to communicate with. The processor $P(j)$ is the *target processor* for $P(i)$, in this step of computation. The goal of each processor is to learn the identification of its target processor. Once this is done, processors may communicate with the targets and simulate the required step of computation, as described in Section 2.

The list is assumed to be cyclic. The distance $dist(i,j)$ from processor $P(i)$ to its target $P(j)$ is the number of links that need to be traversed in order to get from $P(i)$ to $P(j)$. Thus $dist(i,j) = (j-i) \bmod l$, where l is the length of the list.

Communication along the list uses the idea of pointer jumping. We use two kinds of links. The links of the underlying cyclic list are referred to as the *primary links*. They are never modified in the course of computation, once having been established. The links that are doubled are called *secondary*. When the communication starts, the secondary links are initialized to the primary ones. The doubling instruction makes the new link value $L(i)$, for the ith node of the list, equal to $L(L(i))$.

We explain the idea of how doubling is used by the following example. Suppose that just *one* processor $P(i)$ needs to communicate with some other processor $P(j)$, so there is no concurrency involved. Let $d = dist(i,j)$ and let $b_0 \ldots b_{s-1} b_s$ be the binary representation of d, where b_0 is the least significant bit. The value b_k is called the kth *distance bit* of $P(i)$. The distance to the target is $\sum_{k=0}^{s} b_k \cdot 2^k$. Processor $P(i)$ covers this distance in at most s steps. In step 0 it traverses the primary link iff $b_0 = 1$. Further, in the kth step $P(i)$ traverses the appropriate secondary link iff $b_k = 1$. By saying that processor $P(i)$ *traverses* a link pointing to processor $P(j)$ we mean that $P(i)$ scans the shared memory segment assigned to $P(j)$ and copies appropriate information to its own memory segment.

In the case of the CREW model, when simultaneous reads are allowed but the write is exclusive, the above doubling process can be performed directly by all the processors in parallel. The communication process in the CRCW and EREW PRAM models is more complicated. For the CRCW PRAM, it is necessary to consider separately different aspects of simultaneous writes. In the EREW case, we will use dynamic tree-like data structures to avoid read conflicts during the communication along the list.

We start the presentation with the CREW PRAM algorithm. It has the simplest structure and algorithms designed for the other models are its extensions.

4.1 The CREW PRAM

The fault-free cells in the *memory segment* $M(i)$ assigned to processor $P(i)$, for $i = 1, .., l$, are organized as follows. The first m cells are used to store the memory cells of the simulated PRAM. The next three cells contain the primary (static), secondary and target (dynamic) links in the list. We use the notation $P(i).prim$, $P(i).sec$ and $P(i).target$ for the primary, secondary and target links, respectively. Moreover, in all the PRAM models, in every memory segment some additional cells are used in the simulation process. Recall that every processor $P(i)$ has to simulate the work of m processors $p_{(i-1)m+1}, ..., p_{im}$ of the simulated PRAM, as described in Section 2. Every read/write step of the CREW PRAM proceeds in $s = \lceil \log l \rceil$ phases, see the detailed description of the procedure READ/WRITE STEP.

Procedure READ/WRITE STEP;
for $i = 1$ **to** l **in parallel do**
  for $x = 1$ **to** m **do**
    Get index ind of the memory cell affected by the processor $p_{(i-1)m+x}$
      of the simulated PRAM;
    $j := ind$ div l; $z := ind$ mod m; $d := dist(i, j)$;
    $P(i).sec := P(i).prim$; $P(i).target := P(i)$;
    for $k = 1$ **to** s **do**
      $b :=$ the kth bit of d;
      if $b = 1$ **then** $P(i).target := P(P(i).target).sec$;
      $P(i).sec := P(P(i).sec).sec$;
    end_for
    The read/write operation is performed at the zth fault-free cell
      of the memory segment $M(P(i).target)$;
  end_for
end_for.

All the read operations can be performed simultaneously since they are allowed. The only problem occurs when the simultaneous write at the last step of the algorithm is executed. According to the semantic of the CREW PRAM model, if two processors attempt to write to the same memory cell in the same write step then there is a runtime error. Notice that in our simulation the write step is divided into m consecutive substeps, thus some simultaneous writes could remain unrecognized and the computation process would continue, even if there is a runtime error in the simulated computation. To prevent this, every processor $P(i)$ is supported by its (right) neighbor in the list. The neighbor first checks if the destination memory cell of $P(i)$ has been already affected during the current step (one additional bit for every simulated memory cell is needed). If the cell was written to then processor $P(i)$ and its neighbor write their values simultaneously causing a runtime error. We have shown:

Theorem 3 *Every* read/write *step of the simulated CREW PRAM can be performed deterministically in time* $\mathcal{O}(\log n)$. □

4.2 The CRCW PRAM

In the CRCW PRAM model all the read operations are handled according to the scheme of the read/write step introduced in subsection 4.1 but certain aspects of simultaneous writes must be handled more carefully.

During a write step only one value can be written into a given memory cell. In the PRIORITY model, the successful processor is the one with the largest priority (the smallest index) among all processors that want to access the same memory cell. Since the write step is divided into m substeps, the index of the processor that performed a successful (but only tentative) write is stored. In every further write substep, the processor writes its value only if the already stored value has been written by a processor with a smaller priority. In this case we need one more cell, for every simulated memory cell, to store the rank of the processor that has written the value.

In the ARBITRARY model, during a write step at a given memory cell, the value written by an arbitrary processor is good, hence no additional precaution needs to be taken.

The write step in the COMMON model is handled as follows. Every writing processor is supported by its (right) neighbor in the list. The neighbor first reads the already stored value. Then the (writing) processor and its neighbor write their values simultaneously. The computation is terminated if distinct values have been written in the same substep. If that happened in distinct substeps then there is substep x such that the first different value appeared in the xth substep. The only value that has been written there during the first $x-1$ substeps can be read by the neighbor in the list, and in the xth substep a write conflict (runtime error) can be made to happen.

Theorem 4 *Every* read/write *step of the simulated CRCW PRAM can be performed deterministically in time* $\mathcal{O}(\log n)$. □

4.3 The EREW PRAM

Now we show how to perform the read/write step in the model EREW. The main problem that arises is that if many processors have reached a node of the list then simply reading the value of the link would create access conflicts.

Similarly as in the stronger models, the computation proceeds in $s = \lceil \log l \rceil$ phases, where l is the length of the list. The kth phase begins with the kth doubling of the

secondary links. At each phase, say k, there is a set of processors $\mathcal{P}(v,k)$ associated with the list node v: these are the processors traversing the secondary links that are currently at the node v. A phase is performed in three parts. In part one, just after link doubling, the processors in set $\mathcal{P}(v,k)$ are partitioned into two groups (each one possibly empty) depending on the kth distance bit. In part two the processors with the bit equal to 1 are moved to the next node along the appropriate secondary link, those with bit 0 stay at the node. Simultaneously, some new processors arrive at node v. During the third part of the phase, the old and the new processors at a node are merged into a new set $\mathcal{P}(v,k+1)$.

The processors in every set $\mathcal{P}(v,k)$ are organized in a *compressed distance tree* $T(v,k)$ defined below. The *distance tree* is a binary tree whose edges are labeled by ones and zeros. Every branch in a distance tree represents a suffix $d|k = b_k b_{k+1} ... b_s$, where d is the distance of some processor in set $\mathcal{P}(v,k)$ from its target. The distance tree is a prefix tree, that is, its two branches split at their longest common prefix. A *chain* in a distance tree is a maximal sequence of adjacent nodes with outdegree 1. The *compressed distance tree* is a distance tree modified in such a way that all the chains are compressed into single edges. The length of an edge is the number of bits in the corresponding chain. Every edge with the first bit equal to 0 is called a *left edge*, all the other edges are called *right* ones. Similarly, a subtree (child) linked by a left edge is called a *left subtree (child)* and one linked by a right edge is called a *right subtree (child)*. The processors from set $\mathcal{P}(v,k)$ are placed in the internal nodes of the compressed distance tree $T(v,k)$ as follows. Every internal node, except for the root, has exactly one processor placed in it. Every processor $P(i) \in \mathcal{P}(v,k)$ (except, possibly, one at the root) is placed in some internal node r_i in the branch $d_i|k$, where d_i is (the binary representation of) the distance of $P(i)$ from its target. The bottom part of the branch $d_i|k$ always ends at the leftmost leaf in the right subtree of r_i. We say that $P(i)$ *covers* that leaf. The root of degree two has two processors placed in it. One of them covers the leftmost leaf of $T(v,k)$ and the second one covers the leftmost leaf of the right subtree of $T(v,k)$. A processor placed in the root of the compressed distance tree is called the *leader* of the set $\mathcal{P}(v,k)$. If a root has two processors placed in it, then the set $\mathcal{P}(v,k)$ has two leaders. They are called *partners* in this case. Only leaders of the group $\mathcal{P}(v,k)$ have links to the node v in the list, all other processors in this group are involved in holding the structure of the tree.

In phase k, the tree $T(v,k)$ representing the set $\mathcal{P}(v,k)$ is split at the root into two temporary trees if the root has two children. The first tree, the *waiting tree*, contains all the processors that do not use the current secondary link, that is, their kth distance bit is 0. The second tree, called the *traversing tree*, contains all the processors which use the current secondary link. When the tree T is split, only the leader of the set of processors in the traversing tree uses the current secondary link in the list and moves to the new destination u. By the *move* in the traversing process we mean that the value of the link at the leader is changed from v to u. The traversing tree is used in the construction of the distance tree $T(u,k+1)$ for the set $\mathcal{P}(u,k+1)$, where u is

the node of the list reachable from v by the current secondary link. The traversing tree derived from $\mathcal{P}(v,k)$ is merged with the waiting tree derived from $\mathcal{P}(u,k)$.

When the traversing part of phase k is finished, the first bit from every branch is deleted in each traversing and waiting tree. As a result, every leader in the tree either relocated from the root to its only child, also copying the information about the list to its new partner, or deletes the first bit in the edge directed to the child, if the length of the edge is at least two. The updated waiting and traversing trees, which are currently associated with the same node u in the list, are then merged to form a new distance tree $T(u,k+1)$.

The merging process starts from the roots of trees and continues recursively towards the leaves as follows. Let A and B, respectively, be the updated traversing and waiting trees currently associated with the node u in the list. See Figure 1, where operation $\oplus$ denotes the result of merging two trees called their join. We assume that the roots in both trees have two children. Otherwise, at the side (left or right) where one of the subtrees is missing, the merging process is performed by linking the only existing subtree (if any) to the new common root. The merging process is performed in pairs of trees ($A1$, $B1$) and ($A2$, $B2$) independently since it will result in two independent subtrees of $T(u,k+1)$. We will follow the merging process only in one pair, in the other one it is performed analogously. Three cases need to be considered during the recursive merging process (see Figure 1).

Case 1. The edges l_{A1} and l_{B1} differ and none of them is a prefix of the other. In this case a new node is created and the merging process is stopped. The edge l_1 which links the new node and the root of the tree is defined as the longest common prefix of l_{A1} and l_{B1}.

Case 2. One of the edges l_{A1} or l_{B1} is a proper prefix of the other, as in Figure 1, where l_{A1} is a proper prefix of l_{B1}. In this case a new node is created and it is connected with the root by edge l_{A1}. The tree $B1$ is extended at its root by the edge $l_{\tilde{B}1}$ (l_{B1} without its prefix l_{A1}), thus forming the tree $\tilde{B}1$. The merging process between $A1$ and $\tilde{B}1$ is performed recursively.

Case 3. The edges l_{A1} and l_{B1} are the same. In this case the edges l_{A1} and l_{B1} are merged into a single new edge and the roots of trees $A1$ and $B1$ are coalesced into a single new vertex. The merging process between $A1$ and $B1$ is performed recursively.

At any level of the merging process the processors are placed in such a way that they satisfy the general rules for processor positions in the distance tree, as specified earlier in this subsection.

At the root level two processors assigned to the left subtrees choose that one among them which covers the leftmost leaf in their join. This is that processor (like a in Figure 1) whose distance to the target is smaller (actually its distance is smallest in the whole set $\mathcal{P}(u,k+1)$). The other processor (like b in Figure 1) is sent down to a

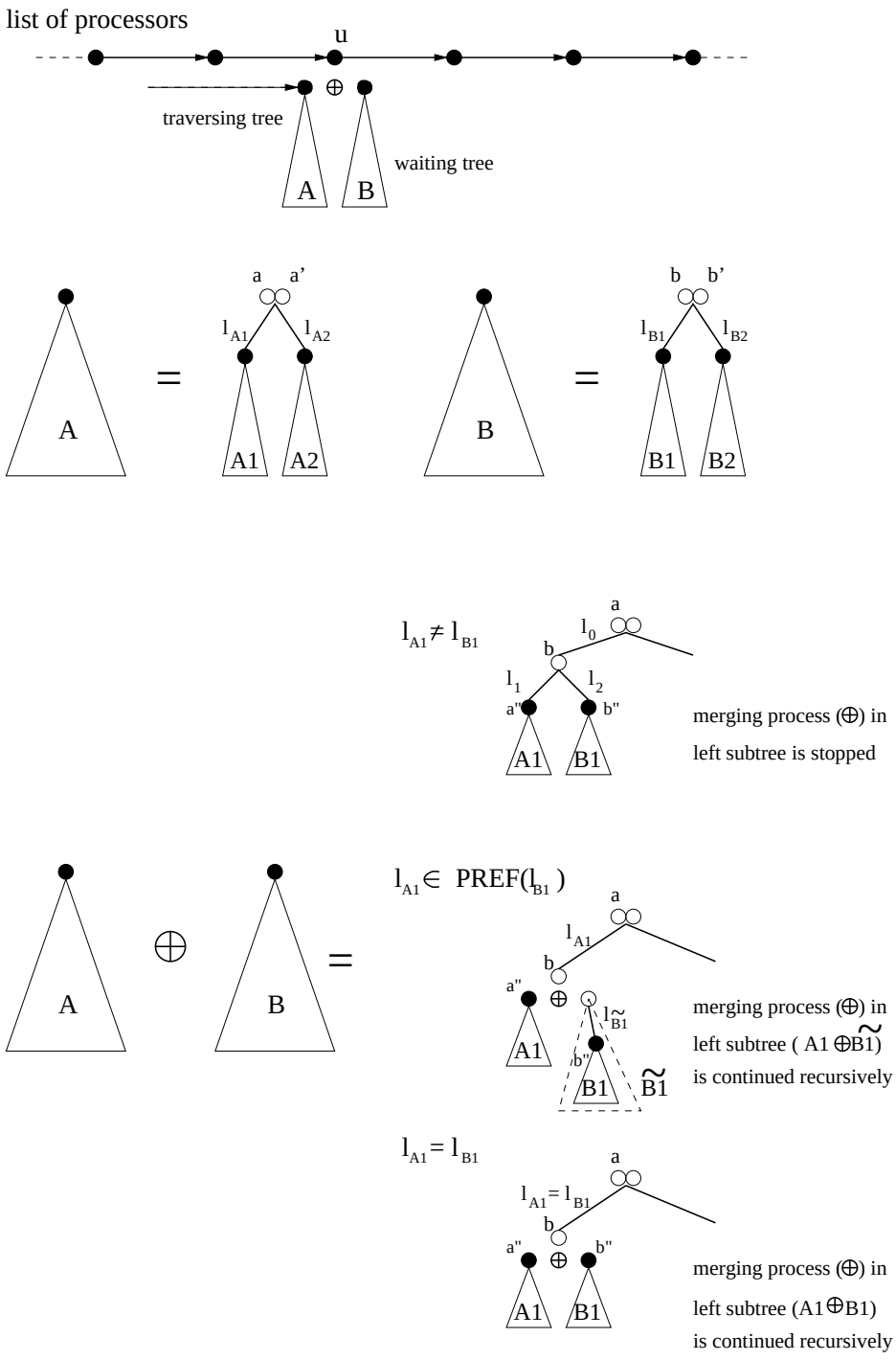

Figure 1: A step of merging.

new vertex (Case 1) or joins the recursive merging process (Case 2 and Case 3).

At the new internal node w the processor which came from the upper level always covers the leftmost leaf of one (say $T1$) of currently merged trees $T1$ and $T2$. The leftmost leaf of the other tree ($T2$) is covered by some processor already placed at the upper level. Processors placed in the roots of both trees choose the one (among them) which covers the leftmost leaf in the join of their right subtrees. The chosen processor is placed at w and the other one is sent down to participate in the merging process of the right subtrees. The processor which came from the upper level is sent down to participate in the merging process of the left subtrees.

The key idea behind the performance of the algorithm is to allow the root (leader) to traverse the secondary links just after the operations needed for splitting and merging the trees regarding it and its children have been completed. To omit synchronization problems one phase of link doubling is associated with two phases of splitting and merging, at odd and even levels separately. Below the root, other processors are still busy reorganizing the tree, but this does not affect the root, hence the latter can follow the current secondary link. Since the operation is done in parallel on many levels, the recursive merging is actually a pipelined operation. If we waited for the whole structure of the tree to be updated, there would be another logarithmic factor in the time performance. We assume that processors have a sufficiently large menu of standard bit operations that allows them to perform the needed operations on strings of bits in time $\mathcal{O}(1)$.

Since there is a logarithmic number of pipelined phases in the communication process, the following theorem holds.

Theorem 5 *Every* read/write *step of the simulated EREW PRAM can be performed deterministically in time* $\mathcal{O}(\log n)$. □

Now theorem 1 follows immediately from theorems 2, 3, 4 and 5.

5 Input for faulty machines

The aspect of input/output operations for a PRAM has never been much discussed, due mainly to the fact that the model itself is theoretical and disregards the real costs of communication between the processors and the memory modules. Usually it is simply assumed that the input has been stored somehow either in the processors' local memories or in the shared memory prior to the start of the computation. We adopt the approach to provide the input directly to the processors. A processor may stay idle (dormant) during the computation because either it is faulty or its assigned segment of shared memory cells does not contain sufficiently many fault-free elements.

The categorizing of processors into active and dormant is done during preprocessing. The question arises, how are the preprocessing and input operations related. It would be hardly acceptable to assume that only *after* preprocessing the input operation starts because that would require a very flexible (and hence complicated and costly) input hardware. Therefore we work under the assumption that the input is provided to all processors regardless of their faults or faults in the shared memory.

There are at least $\alpha \cdot n$ active processors arranged in a list after preprocessing. We want to provide the whole input to the first $\alpha \cdot n$ processors in the list, by resorting to the general input mechanism which distributes the input among all the n processors. A possible solution is to encode the input, in such a way that it can be retrieved from the fraction of information known to the active processors.

Methods of encoding and then retrieving the original message in a situation when only a part of the transmitted information is available are known as error-correcting codes. We apply the specific simple and fast method popularized by Rabin [13] under the name of *information dispersal*.

There are $u_1, \ldots, u_{\alpha n}$ original input strings, each comprised of some t bits; it is required that $2^t \geq n$. They are encoded as a sequence $v_1, \ldots, v_n$, where $v_i = g(\omega^i)$ and g is the polynomial

$$g(x) = u_1 + u_2 x + \ldots + u_{\alpha n} x^{\alpha n - 1}$$

The arithmetic is in the field $GF(2^t)$. The element ω is a 2^tth primitive root of unity in $GF(2^t)$. The task of encoding is equivalent to computing the Discrete Fourier Transform and can be implemented efficiently by the FFT algorithm. Suppose that $v_{i_1}, \ldots, v_{i_{\alpha n}}$ are the strings stored by the active processors in the list (in any order). This gives $\alpha \cdot n$ values at distinct points of a polynomial of degree $\alpha \cdot n - 1$. The task of retrieving the input $u_1, \ldots, u_{\alpha n}$ is equivalent to obtaining the coefficients of a polynomial from its values, that is, to *interpolating* the polynomial. There is an algorithm for this problem that runs in time $\mathcal{O}(\log^3 n)$ on an EREW PRAM with n processors (see [5] and the references therein). The algorithm computes the Lagrange interpolation formula and is reduced to polynomial evaluation at $\mathcal{O}(n)$ points and then to the FFT algorithm. This algorithm specialized to our problem can be implemented to run faster, even on the butterfly (see [11] for the description and properties of the butterfly). By a *normal butterfly algorithm* we mean an algorithm in which a step of computation is performed by the nodes on one level, and the consecutive levels are used in a cyclic fashion.

Lemma 2 *The decoding of information dispersal from the fraction of $\alpha \cdot n$ strings can be performed on the $\mathcal{O}(\log n)$-dimensional butterfly in time $\mathcal{O}(\log^2 n)$ by a normal algorithm.*

Proof. The general interpolation algorithm resorts to the algorithm of evaluating a polynomial at $\mathcal{O}(n)$ points, which runs in time $\mathcal{O}(\log^2 n)$. This algorithm can

be replaced by the algorithm to evaluate polynomials at the powers of a root of unity, because the arithmetic is in a finite field. Notice that we need to evaluate the polynomials at some bounded fraction of *all* the elements of the field, which can be performed in time $\mathcal{O}(\log n)$ by the FFT algorithm; in this way we gain the $\log n$-factor. The communication of processors is that needed for the FFT algorithm and of a full-binary-tree pattern, and can be implemented on the butterfly as a normal algorithm. □

We can adapt this algorithm, due to the following:

Lemma 3 *A normal butterfly algorithm can be implemented on a list of active processors with delay $\mathcal{O}(1)$, provided the size of the list is at least equal to the number of nodes in one level of the butterfly.*

Proof. We need a two-directional cyclic list of length equal exactly to the size of a level of the butterfly, but this can be assumed to have been taken care of during preprocessing. Each processor simulates a row of the butterfly. The connections to other rows can be obtained dynamically by doubling the list links in both directions. A processor chooses the particular link depending on the bit representation of the row number. □

After decoding, the input strings are stored in the order corresponding to physical ordering of processors and not in the order of the list. They can be rearranged by one application of the operation of communication along the list. As a conclusion of this discussion we obtain the following result:

Theorem 6 *The input encoded by the information dispersal method can be retrieved by the active processors of an EREW PRAM in time $\mathcal{O}(\log^2 n)$.* □

6 Conclusions

We presented a deterministic simulation of the fully operational PRAM on a PRAM with faulty processors and faulty memory. The simulation is fast, but its optimality is an open problem. In particular, investigating the specific problem of building a list of operational processors in a deterministic way is interesting. We showed how this can be done in time $\mathcal{O}(\log^2 n)$, but we know no lower bound better than the trivial $\Omega(\log n)$.

References

[1] B.S. Chlebus, A. Gambin, and P. Indyk, PRAM computations resilient to memory faults, in *Proceedings of the 2nd Annual European Symposium on Algorithms,* ed. J. van Leeuwen, Utrecht, The Netherlands, 1994, Springer LNCS 855, pp. 401–412.

[2] B.S. Chlebus, L. Gąsieniec, and A. Pelc, Fast deterministic simulation of computations on faulty parallel machines, in *Proceedings of the 3rd Annual European Symposium on Algorithms,* 1995, to appear.

[3] K. Diks and A. Pelc, Reliable computations on faulty EREW PRAM, *Theoretical Computer Science*, to appear.

[4] A. Gibbons, and W. Rytter, *"Efficient Parallel Algorithms,"* Cambridge University Press, 1988.

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

[6] P.C. Kanellakis, and A.A. Shvartsman, Efficient parallel algorithms can be made robust, *Distributed Computing*, 5 (1992) 201-217.

[7] P.C. Kanellakis, and A.A. Shvartsman, Efficient parallel algorithms on restartable fail-stop processors, in *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing,* 1991, pp. 23–36.

[8] R.M. Karp, and V. Ramachandran, Parallel algorithms for shared-memory machines, in *"Handbook of Theoretical Computer Science,"* ed. J. van Leeuwen, Elsevier, 1990, vol. A, pp. 869–941.

[9] Z. M. Kedem, K. V. Palem, A. Raghunathan, and P. Spirakis, Combining tentative and definite executions for very fast dependable parallel computing, in *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing,* 1991, pp. 381-390.

[10] Z. M. Kedem, K. V. Palem, and P. Spirakis, Efficient robust parallel computations, in *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing,* 1990, pp. 138-148.

[11] F.T. Leighton, *"Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes,"* Morgan Kaufman Publishers, San Mateo, California, 1991.

[12] F. Meyer auf der Heide, Hashing strategies for simulating shared memory on distributed memory machines, in *Proceedings of the 1st Heinz Nixdorf Symposium "Parallel Architectures and their Efficient Use,"* ed. F. Meyer auf der Heide, B.

Monien, A.L. Rosenberg, Paderborn, Germany, 1992, Springer LNCS 678, pp. 20–29.

[13] M.O. Rabin, Efficient dispersal of information for security, load balancing, and fault tolerance, *Journal of ACM*, 36 (1989), 335–348.

[14] A. A. Shvartsman, Achieving optimal CRCW PRAM fault-tolerance, *Information Processing Letters,* 39 (1991) 59-66.

[15] L.G. Valiant, General Purpose Parallel Architectures, in *"Handbook of Theoretical Computer Science,"* J. van Leeuwen (Ed.), Elsevier, 1990, vol. A, pp. 869–941.