Computing in Fault Tolerant Broadcast Networks and Noisy Decision Trees [*]

Ilan Newman
University of Haifa, Haifa, Israel
`ilan@cs.haifa.ac.il`

Abstract

We consider a fault tolerant broadcast network of n processors each holding one bit of information. The goal is to compute a given Boolean function on the n bits. In each step, a processor may broadcast one bit of information. Each listening processor receives the bit that was broadcast with error probability bounded by a fixed constant ϵ. The errors in different steps, as well as for different receiving processors in the same step, are mutually independent. The protocols that are considered in this model are *oblivious* protocols: At each step, the processors that broadcast are fixed in advanced and independent of the input and the outcome of previous steps.

We present here the first linear complexity protocols for several classes of Boolean functions. This answer an open question of Yao [28], considering this fault tolerant model that was introduced by El Gamal [4] and studied also by Gallager [10].

Key words: 'Fault tolerant computations', 'Noisy decision trees', 'Gallager's broadcast problem'

[*]a preliminary version of this work appeared in IEEE Conference on Computational Complexity 2004: 113-122.

1 Introduction

In a *fault tolerant* scenario the outcome of any single operation is erroneous with probability bounded by some constant ϵ. It is also assumed that errors for different operations are independent. Given a complex algorithm, one often wishes to get a noise-immune algorithm, namely one that ends with an overall constant error probability. If the outcome of each operation is Boolean, and there are total of t operations, then repeating each operation for $O(\log t)$ times and taking majority as the outcome, typically results in a noise-immune algorithm at the cost of $O(t \log t)$ operations. It would be desirable to construct noise-immune algorithms paying less overhead. This fault tolerant scenario was extensively studied for different computational models. Such work includes [19, 3, 20, 9, 21] for Boolean circuits with noisy gates, [7, 21, 5, 15, 14] for noisy decision trees, [23, 22] for communication models and others [25, 17, 8, 24, 2]. It turns out that for some models it is impossible to do better than the overhead factor of $O(\log t)$ and sometimes there is a non-trivial better way.

El Gamal [4] formulated the following *noisy* broadcast network model, typical for some radio networks. The system is composed of n processors using broadcasts as the means of communication. At each step of a protocol, each processor may broadcast a bit, while every processor receives all the bits transmitted knowing the source of each. Every single bit received, at each processor, might be erroneous (in its value) with a probability bounded from above by an a priori known constant ϵ. Furthermore, it is assumed that the errors for different broadcasts, as well as for different receivers in the same broadcast, are independent. At the beginning of the operation, each processor has one bit. The goal is to compute a given Boolean function on all the bits. The complexity of the protocol is the total number of broadcasts. The protocols that are considered are *oblivious*, that is, whether at a step i processor P_j broadcasts or not may not depend on its initial bit or on what he has received so far. It may only depend on n, i and j. The content of the transmission may depend on the inputs. The reason for considering oblivious protocols is both practical, as a model of certain radio networks, as well as the natural way to avoid situations in which information is inferred from whether a certain P_i has broadcast or not, rather than from the content. Note that for non-oblivious protocol, all the bits can be recovered in a single round of $O(n)$ broadcasts in which each processor that holds a '1' broadcasts.

Gallager [10] showed that the parity functions can be computed in $O(n \log \log n)$ broadcasts which was the first improvement on the trivial $O(n \log n)$ solution. His protocol ends with all processors knowing all bits, hence implying the same bound for every Boolean function. Yao [28] posed it as an open question whether the OR function (or other non-trivial functions) can be computed in $O(n)$ broadcasts. Very recently Goyal et al. [13] showed that in order to recover all bits any protocol needs $\Omega(n \log \log n)$ broadcasts. However, it is still open whether there is any Boolean function whose computation requires more than linear number of broadcasts. For the specific OR function, Feige and Kilian [6] constructed an $O(n \log^* n)$ protocol that has the additional feature of taking $O(\log^* n)$ rounds.

Other related models and results: A related, but more restricted model than the fault tolerant model, which we call the 'statistical model' assumes that the error probability of each operation is *exactly* ϵ. Namely, there is fixed constant ϵ (known or unknown to the protocol) so that for each pair of processors (P_i, P_j), the error of P_i when receiving a transmission from P_j is *exactly* ϵ. In particular, in such a model, the parameter ϵ can be estimated, even if unknown, to any predefined precision, by sampling. A protocol, designed for such model for the fixed parameter ϵ, is not required to work for the situation where ϵ decreases and, in particular, if it becomes 0. In contrast, in the fault tolerant model the error distribution is not known. There is only a guarantee that the error per operation is *at most* ϵ and that errors are independent for different operations. Any adversarial probability distribution that meets those limitation is a legitimate one. Hence, a fault tolerant algorithm designed for a parameter ϵ should certainly work within its defined complexity and error bound, for the situation in which the actual error is bounded by any $\delta < \epsilon$, and in particular, for the errorless case.[1] We note that Feige and Kilian, [6], constructed their protocol for a somewhat stronger model than the fault tolerant. Roughly, the noise is as in the statistical model, but at each round (or step) an omniscient adversary can revert any erroneous message. Our algorithms can be seen to be suitable for this model, however, this will not be discussed further.

Kushilevitz and Mansour [16] constructed a linear complexity protocol for computing threshold functions in the statistical noisy broadcast model. Their algorithms are not correct for the fault tolerant model. Goyal et al. [13], proved their lower bounds for the statistical model (and hence the lower bound applies to the fault tolerant model as well).

Our Results: We answer in the affirmative Yao's question: We construct here several $O(n)$ fault-tolerant protocols in the standard noisy broadcast model for various classes of Boolean functions including the $\mathrm{OR}, \mathrm{AND}$, functions that have $O(1)$ size 1-witnesses (0-witnesses), functions that have linear size AC_0 formulae and some other functions. Thus, we give the first linear complexity protocols for both the OR function, and some other interesting non-trivial classes of Boolean functions. We also show how to find the whole input word using $O(n \log r)$ broadcasts, provided that the number of 1's in the input is at most r. Note that for r such that $\log r = o(\log \log n)$ this is better than Gallager's bound. We also treat briefly the issue of the number of rounds needed for the OR function. We construct a $O(n)$ complexity fault-tolerant decision tree algorithm of $O(\log^* n)$ rounds and show how to implement a variant of it as a $O(\log^* n)$-rounds protocol in the broadcast model.

Our results are based on two main ingredients. The most important one is a reduction of the task of computing Boolean functions in the broadcast network model to a certain efficient fault tolerant computation in the noisy decision tree model. In order to implement fault-tolerant decision tree algorithms in the broadcast model we introduce a property that measures the adaptiveness of such algorithms, denoted here as the *parallel time.* We construct new algorithms for some classes of Boolean functions for the noisy decision tree model. In particular, we construct two linear algorithms for the OR, with one being simultaneously of total linear complexity and $O(\log^* n)$ rounds ($O(\log^* n)$ parallel time). The second ingredient is a simple application of some ideas of parallel computing.

The rest of the paper is organized as follows: In Section 2.1 we define the model and observe some basic properties. In Section 2.2 we define the noisy decision tree model and

[1]El-Gamal and Gallager worked in the information theoretic model and may not have considered the fault tolerant one. However, Gallager's algorithm is fault tolerant.

present an $O(n)$ fault tolerant algorithm for the OR. In Section 3 we describe a linear time protocol for the OR function in the fault tolerant broadcast model. In Section 4 we discuss the general issue of simulating fault tolerant decision trees by protocols in the broadcast model, then in Section 5, we develop fault tolerant decision tree algorithms which can be simulated in broadcast model for several classes of Boolean functions . In particular, to do this we generalize the decision tree complexity to the case in which every variable has its own query-cost and construct a fault-tolerant algorithm which is linear in the total cost for the OR. Then, in Section 6 we discuss the all-bit word problem in which the outcome should be that every processor knows the whole input word. In Section 7 we discuss briefly algorithms for OR that are very efficient in terms of the total number of rounds, in both the decision tree model and the broadcast model. Finally, in Section 8 we present some open problems.

2 Preliminaries

2.1 The model

Let $\{P_1, ..., P_n\}$ be a set of n processors. An ϵ-fault tolerant protocol in the noisy broadcast model is a sequence of m rounds; in every round $j = 1, ..., m$, a certain subset $S_j \subseteq [n]$ of the processors broadcast, one bit each. For each processor $P \in S_j$, every processor $Q,\ Q \neq P$ holds a random variable $X^j_{P,Q}$ that equals the bit that was broadcast by P with probability at least $1 - \epsilon$. We say that ϵ in this case is the bound on the *communication-error*. In addition, the variables $X^j_{P,Q}, j = 1, ..., m,\ P \in S_j,\ Q \in [n] - \{P\}$ are independent. Furthermore, the subset S_j, namely the identity of the processors that broadcast at round j is predefined in the protocol and is independent of the input or previous information that processors might obtain in the rounds previous to j. Note that the value of the bit that is transmitted by P at round j might depend on the input as well as the previous information received by P. This restriction of being *oblivious* is to avoid trivialities such as the following one round protocol: Each processor broadcasts if and only if it has a '1'. Then all input bits become known to each processor, regardless of any error, just by detecting which processors have broadcast. In the rest of paper we refer only to oblivious protocols.

The complexity of an oblivious protocol is the total number of broadcasts, namely $\Sigma^m_{j=1}|S_j|$. Another complexity parameter that is of (secondary) interest is m, the number of rounds of the protocol. A protocol is said to compute a (partial) function $f()$ if for any allowed input x, it ends with a value f_i at each processor $i = 1, ..., n$, such that Prob$[\exists i\ f(x) \neq f_i] \leq 1/3$.

As already discussed in Section 1, ϵ, the bound on the error probability is a priori known. It is perfectly legitimate for an alleged adversary to corrupt each, say, odd round j with probability $\delta_j < \epsilon$ while letting each even round to result in an error-less reception (or alternatively, decide for each processor R and each round j to corrupt $X^j_{R,P}$ at the jth round with probability $p_{r,j} < \epsilon$).

Any protocol can be serialized keeping the same number of broadcasts. We will some times do this in order to refer to some particular order of the broadcasts within a round.

In this case we talk about steps in which a single processor broadcasts.

Finally since simple amplification can be done in this model, the exact value of ϵ, the communication-error bound, is not important. Also, for the same reason we could make the final error probability bound to be any constant $c < 1/2$ (instead of $1/3$) keeping the same asymptotic complexity. Thus we say that a protocol is *fault-tolerant* in this model if there are two constants $\epsilon, \delta < 1/2$ (and independent of the input length n) such that under the assumption that the communication-error bound is at most ϵ, the total error probability of the algorithm is at most δ.

An important 'gadget' that we recurrently use is the following computational task. Suppose that a fixed known processor P holds a word $w \in \{0,1\}^k$ that he wishes to make known to every other processor. We denote this task as the *faithful distribution* of w. The following simple observation states that 'faithful distribution' can be done efficiently.

Observation 2.1 *Faithful distribution of $w \in \{0,1\}^k$ to n processors takes $O(max\{k, \log n\})$ broadcasts. As a result each processor knows w correctly with probability $1 - 1/poly(n)$.*

Proof: Suppose first that $k \geq 16 \log n$. Let us fix a linear size error correcting code that corrects up to 1/3-fraction of errors (the fact that such codes exists and can be efficiently constructed is known see e.g [18, 11, 26]). P_1 encodes w using such a fixed code into a word $c(w)$ of size $C(k) = O(k)$ and broadcasts each bit of $c(w)$. Each P_i, $i = 1, ..., n$ decodes w from the received word. P_i erroneously decodes w only if more than $C(k)/3$ errors occurred. By Chernoff bound [1], this will occur with probability at most $exp(-2(1/3 - \epsilon)^2 C(k))$. For $\epsilon < 1/12$, this would be less than $1/n^2$ (or below any $1/poly(n)$ for suitable ϵ). This implies that with probability at least $1 - 1/n$ all processors have decoded w correctly. If $k < 16 \log n$ we append w to itself $\lceil 16 \log n/k \rceil$ times, to form a word of size k' where $16 \log n \leq k' < 32 \log n$. We then faithfully distribute it using $O(\log n)$ broadcasts. ■

2.2 Noisy Boolean Decision-Trees

A major ingredient in what follows are fault tolerant algorithms for noisy Boolean decision-trees. Several such models were discussed in the literature. The relevant one to this work is the seminal work of Feige et al. [7]. In this model, a given Boolean function on n variables is to be computed on an unknown input using queries to the variables. As a standard fault tolerant model, for each variable queried, an erroneous answer is received with probability bounded from above by an a priori bound ϵ and independent of any other queries or answers. Hence, an algorithm is a decision tree. In each step a variable is queried, then, based on the answer a next variable is being queried and so on, until a final decision is made. The complexity of such algorithm is the maximum number of queries for the worst case input. Namely, it is the depth of the tree. A tree $\mathcal{T}$ computes a (partial) function if, for every relevant input, its error probability is bounded by $1/3$. Again, as standard amplification is possible, we may replace the $1/3$, as well as ϵ, with any other constants that are bounded away form $1/2$. Thus, here too, we say that a decision tree is *fault-tolerant* if there exist two constants $\epsilon, \delta < 1/2$, such that if the query error is at most ϵ then the output error is at most δ.

An important parameter of complexity, although non-standard for decision trees, is the *amount of adaptiveness,* which can be measured by the 'parallel time' of the algorithm. Namely, for each input, the algorithm defines a certain path in the tree in which a sequence of variables are probed one after the other. However, in many cases, this order is arbitrary in the sense that the query at step j does not depend on the answer at step $j-1$ for some computational path. In such a case, the queries of step $j-1$ and step j could have been asked in parallel. If the algorithm is presented as a decision tree, then there is no explicit parallelism. However, as we care about this extra parameter, the algorithms will be described in terms of rounds: In each round a *multiset* of variables are queried. The complexity of the algorithm is the total number of queries, while the parallel time is the number of rounds for the worst case run. Feige et al. proved (among other things):

Theorem 1 *[7] There is a fault tolerant decision tree for the* OR *function on n variables of complexity $O(n)$ and parallel time $O(\log n)$.*

In Section 7 we construct a variant of this algorithm that is simultaneously of $O(n)$ total complexity, while it takes only $O(\log^* n)$ rounds. The following is, however, simpler, (with a much simpler analysis than of [7]), and will turn more 'handy' to simulate as a protocol for the noisy broadcast model.

Observation 2.2 *There is a fault tolerant decision tree $\mathcal{T}$, that for any input $x \in \{0,1\}^n$ outputs an index $i \in [n]$ which is the minimal for which $x_i = 1$, if such i exists, or an arbitrary i if $\mathrm{OR}(x) = 0$. For every $\epsilon < 1/2$, if the query error is guaranteed to be at most ϵ then the failure probability is also bounded by ϵ. The complexity of $\mathcal{T}$ is $O(n)$ and its parallel time is $O(\log n)$.*

Before we prove Observation 2.2 we introduce the following notation: For a multiset $S = (i_1, \ldots, i_k)$ (which will always be assumed to be ordered) we let $|S| = k$, namely the size of the multiset is the number of members in it counting multiplicities. For a multiset $S = (i_1, \ldots, i_k)$ we denote by Maj(S) the element α that appears at least $(k+1)/2$ times in S. If there is no such α we set $\mathrm{Maj}(S) = i_1$.

Proof of Observation 2.2:
We assume that the query error is bounded by $\epsilon < 1/64$ (otherwise we use amplification to arrive to this situation). We then prove that the output of the algorithm will be incorrect with probability bounded by 8ϵ. The error can then be reduced to ϵ by standard amplification. We also assume that $n = 4^k$ (otherwise let n' be the smallest integer such that $n' = 4^k$ and $n' \geq n$. We then work with n' instead of n and set $x_i = 0$ for $n < i \leq n'$). Let $m = n/4$, the algorithm $\mathcal{T}$ will be recursive.

First every variable $x_i,\ i = 1, ..., n$, is queried. Let $z_i,\ i = 1, ..., n$ be the corresponding answers.

If $n \leq 4$ then the output is chosen to be the smallest i for which $z_i = 1$, if there is such an i, or $i = 1$ otherwise.

If $n > 4$, $[n]$ is partitioned into m disjoint subsets of size 4, denoted by blocks. In each block containing variables $x_j, ..., x_{j+3}$ an index i is chosen as follows: If all values are 0's then j is chosen. Otherwise, the minimal $i \in \{j, ..., j+3\}$ for which $z_i = 1$ is chosen.

Let $I \subseteq [n]$ be the set of m values thus chosen. Then $\mathcal{T}$ is applied recursively on I for 3 independent times resulting in a multiset (i_1, i_2, i_3). The output is set to be $\mathrm{Maj}(i_1, i_2, i_3)$.

Clearly the total complexity of $\mathcal{T}$ on n variables, denoted by $a(n)$ is bounded by $a(n) \leq n + 3a(n/4)$ which implies that $a(n) \leq 4n$. It is also clear that the parallel time is $O(\log n)$.

Let ϵ_n be the error of the algorithm on n variables. We show inductively that $\epsilon_n \leq 8\epsilon$ (which is easily checked for $n \leq 4$). To analyze the error probability note that $\mathcal{T}$ can't err if $\mathrm{OR}(x_1, ..., x_n) = 0$. Assume then that $\mathrm{OR}(x_1, ..., x_n) = 1$ and let i be the minimum index for which $x_i = 1$. Then $\mathcal{T}$ errs if $i \notin I$, or if $i \neq \mathrm{Maj}(i_1, i_2, i_3)$. The probability of the first event is clearly upper-bounded by 4ϵ, as for i not to belong to I, at least one of the four values in the block that contains i must be wrong. The probability of the second event is upper-bounded by $3\epsilon_{n/4}^2$ as at least two of (i_1, i_2, i_3) must be wrong if $i \neq \mathrm{Maj}(i_1, i_2, i_3)$. Our assumption that $\epsilon < 1/64$ together with the induction hypothesis imply that $3\epsilon_{n/4}^2 \leq 3\epsilon$, hence the probability that $\mathcal{T}$ fails is bounded by $4\epsilon + 3\epsilon < 8\epsilon$ as claimed. ■

Remarks:

1. The algorithm is described as a recursive algorithm. It will be convenient now to unfold the recursion and think of it as running in t rounds for some fixed $t = O(\log n)$. Each round $i = 1, ..., t$ is associated with a multiset S_i of the variables that are queried in the ith round. Note that S_i might depend on the answers to the previous queries. However, note that the sizes $|S_i|$, $i = 1, ..., t$, are fixed and independent of the input or the current run.

2. Once the index i is found, as asserted by Observation 2.2, x_i can be queried for $O(\log n)$ times, enough to verify the value of x_i, which is also the value of $\mathrm{OR}(x)$, with probability at least $1 - \frac{1}{n^2}$.

3 A fault tolerant protocol for the OR in the noisy broadcast model

Our aim is to simulate the algorithm $\mathcal{T}$ of Observation 2.2 and find the first index of a variable that is '1' (if such exists) in linear number of broadcasts. It will now become evident why the 'limited adaptiveness' is crucial. We note that the algorithm of [6] which takes $O(n \log^* n)$ broadcasts and $O(\log^* n)$ rounds, also finds the first processor that holds a '1' (if exists). Here we construct a protocol with $O(n)$ broadcasts. In Section 7 we indicate how to obtain a simultaneous $O(n)$ complexity, $O(\log^* n)$-rounds protocol.

Theorem 2 *Let $P_1, ..., P_n$ be n processors, each holding a Boolean value b_i, $i = 1, ..., n$. There is a fault tolerant protocol of complexity $O(n)$, that ends with P_1 holding an index $r \in [n]$. If $\mathrm{OR}(b_1, ..., b_n) = 1$ then r is the minimal index for which $b_r = 1$, otherwise r is arbitrary. If the broadcast error is bounded by $\epsilon < 1/128$ then the probability that the protocol fails is at most $1/8$.*

Proof: Let $\mathcal{T}$ be the algorithm of Observation 2.2. Following the first remark at the end of Section 2.2, let $S_i \subseteq [n],\ i = 1, \ldots, t = O(\log n)$, be the multisets of variables that are queried at round i. Recall that S_i might depend on the answers of previous queries to variables in $S_{i-1}, ..., S_1$, however, its size $|S_i|$ is known and fixed for all runs. Also, note that $\Sigma_{i=1}^t |S_i| \leq 4n$ (recall that $|S_i|$ is the size of the multiset counting multiplicities). Let $\epsilon < 1/128$ be the bound on the error probability in each broadcast.

The top level idea is the following: Assume that we have an extra processor P_0 whose role would be to 'run' the decision tree $\mathcal{T}$. Namely, P_0 will gather all the answers to the queries done by $\mathcal{T}$ and at the end will hold the answer. In reality, as such P_0 is not available, its actions will be simulated by P_1 in a straightforward way.

We first describe a *non-oblivious* protocol to simulate $\mathcal{T}$. We then explain the details of the oblivious protocol. To simulate the first round, P_0 needs to query the variables in S_1. As $\mathcal{T}$ is known to all processors, this can be done by letting each $P_i,\ i \in S_1$ broadcast its value. At this point P_0 has successfully simulated the first round of $\mathcal{T}$. However, as S_2 depends on the values received for the queries of S_1, in order to carry out the second round in a similar way, each P_i needs to know the values that P_0 obtained in the first round of queries. This poses no real problem as P_0 can faithfully distribute his word of answers using $O(max\{|S_1|, \log n\})$ broadcasts, as described in Section 2.1. At this point the second round of $\mathcal{T}$ can be simulated as before and so on. Thus the whole algorithm will work in $t = O(\log n)$ phases, where phase i simulates the i-th round of $\mathcal{T}$. Each phase is composed of two sub-phases, in the first sub-phase every processor in the multiset $\mathcal{P}_i = \{P_i |\ i \in S_i\}$ broadcasts its value. In the second sub-phase P_0 faithfully distributes the 'answers' it heard in the first sub-phase using $O(\max\{S_i, \log n\})$ broadcasts. As $\Sigma|S_i| \leq 4n$ and the number of rounds $t = O(\log n)$, the total number of broadcasts is upper-bounded by $O(\Sigma_{i=1}^t\ max(|S_i|, \log n)) = O(n + \log^2 n) = O(n)$. However, this protocol is *not oblivious;* S_2 depends on the answers to the the queries in S_1 and hence the processors that need to broadcast in the second phase are not known in advance. This, of course, is true for every phase but the first.

It is important to note that in phase i, P_0 does not specify S_i itself but rather just the answers to the queries in S_{i-1}. As the tree $\mathcal{T}$ is known to all processors, this is sufficient to uniquely determine S_i. Hence, after the $(i-1)$th round was simulated, P_0 needs to faithfully distribute only $|S_{i-1}|$ bits of information in order to make S_i known to all processors.

To solve the problem of obliviousness we assume in what follows the existence of additional $4n$ processors, $H_1, ..., H_{4n}$, denoted as helpers. The helpers do not have any values at the beginning of the protocol. We will argue later that those $4n$ helpers can be simulated by the real processors at no increase in the asymptotic cost.

We now get rid of the non-obliviousness as follows. We start by letting each processor $P_i,\ i = 1, ..., n$ broadcast its bit. As a result each helper H_j contains a 'noisy copy' $Z_j(i)$ of the bit of P_i for every i. From this point on, the original processors $P_1, ..., P_n$ do not participate in the protocol, only P_0 and the helpers are responsible for the rest of the work. Recall that $\mathcal{T}$ uses $q \leq 4n$ queries, divided into t multisets. We can serialize the protocol and enumerate the queries as $Q_1, ..., Q_q$. Helper $H_j,\ j = 1, ..., 4n$, will be used just once during the whole protocol - to simulate the answer to the jth query, Q_j, done by $\mathcal{T}$.

Formally, the oblivious protocol works in $t+1$ phases: In phase '0' each $P_i,\ i=1,\dots,n$ broadcasts its bit. As a result every helper H_j holds a 'noisy copy' $Z_j(i),\ i=1,\dots,n$. Let $s_0=0$ and $s_i=\Sigma_{j=1}^{i-1}|S_j|$, then, in phase $i\geq 1$, S_i is known by induction, and the helper $H_{s_{i-1}+j},\ j=1,\dots,|S_i|$, broadcasts the value for the jth query in S_i, which it has due to phase '0'. At this point P_0 gets the answers and faithfully distribute them as described before. This fully determines S_{i+1} which allows for the simulation of the next round.

Note that although the sets $S_i,\ i=1,...,t$, are not known in advance, $|S_i|$ is known and hence the helpers that broadcast at the ith phase are determined in advanced. Thus the algorithm is oblivious.

Since we do not have the additional $4n$ helpers, then for every $i=1,...,n,\ P_i$ simulates helpers $\{H_j|\ j=4(i-1)+1,...,4(i-1)+4\}$. To do this we need to repeat phase '0' for 4 times as we need that each helper will obtain an independent copy of each input variable. Namely, each of the 4 simulated helpers within a processor will use the value of a different and independent copy of the 4 repetitions of phase '0'.

Finally, to analyze the error probability of the protocol note that from the point of view of P_0, the answers it gets are consistent with a simulation of the noisy decision tree with query error bounded by $2/128$. This is true as a value it hears from an helper might be wrong due to the helper broadcast, or might be erroneous due to an error in the value that the helper heard from the corresponding processor at phases '0'. In addition, the values heard by P_0 from different helpers are completely independent. Another source of error is the event that at least one of the faithful distributions fails. By Observation 2.1, this occurs with probability smaller than $1/n^2$ for any specific faithful distribution. Hence, all rounds of faithful distribution are correct with probability at least $1-o(1)$. This implies that the total error is at most $1/8+o(1)$. ■

Remark: We note that using helpers in the proof above works as the errors resulting from the broadcasts of different helpers are independent. However, this is true as P_0 is the sole one to use this information. If more than one listener is using the information of any one helper, all listeners might get the wrong value with constant probability as the helper might have a faulty value.

The protocol above does not fully solve the OR function as only P_0 knows an index r as asserted by Theorem 2. It is quite simple to construct from it a protocol for the OR, which in addition, produces a 1-witness in case the answer is '1'.

Theorem 3 *There is an $O(n)$-complexity protocol that ends with each processor holding an index $i\in[n]$ and such that with probability $1-1/n^2$ all processors hold the same index i. If $\mathrm{OR}(x_1,\dots,x_n)=0$ then with probability $1-1/n^2$ every processor knows that the answer is 0. If $\mathrm{OR}(x_1,\dots,x_n)=1$ then with probability at least $2/3$, i is the smallest such that $x_i=1$ and every processor knows the value of x_i with probability at least $1-1/n^2$.*

Proof: First, the protocol of Theorem 2 is performed, which ends with P_1 knowing an index r as required. Now, P_1 faithfully distributes the index r and then lets each processor broadcast its value (this is just for the sake of being oblivious - the goal here is to get P_r

to broadcast). Then in the following and final round each processor broadcasts the value it heard from P_r and takes the majority of the values it receives in this round as the answer to the OR. $\blacksquare$

Clearly, Theorem 3 implies an analogous result for the AND function.

4 A general simulation Theorem

The simulation of a decision tree algorithm by an oblivious broadcast protocol is quite general.

Theorem 4 *Let $f : \{0,1\}^n \longrightarrow R$ be a (partial) function for which there is a fault tolerant decision tree of complexity $C(n)$ and parallel time $t \leq C(n)/\log n$ then f can be computed by a fault tolerant broadcast protocol of complexity $O(C(n))$.*

Proof of Theorem 4: Let $\mathcal{T}$ be a fault tolerant decision tree for f of complexity $C(n)$ and parallel time $t \leq C(n)/\log n$. We first transform $\mathcal{T}$ into a decision tree $\mathcal{T}'$ of complexity $O(C(n))$, in which exactly $16 \log n$ queries are being made in every round. Then we simulate $\mathcal{T}'$ by an oblivious protocol along the lines of the proof of Theorem 2.

To do this we call a round of $\mathcal{T}$ in which more than $16 \log n$ queries are asked *large*. We partition the queries of each large round into subsets of size exactly $16 \log n$, possibly except for one, which is of size less than $16 \log n$. We then ask the queries in those subsets in consecutive rounds. This results in an algorithm $\mathcal{T}''$ in which in every round there are either $16 \log n$ queries or less. The number of queries in $\mathcal{T}''$ is identical to that of $\mathcal{T}$. It is also easy to see that the parallel time of $\mathcal{T}''$ is $O(C(n)/\log n)$. Now we transform $\mathcal{T}''$ into $\mathcal{T}'$: we add arbitrary queries to every round in which less than $16 \log n$ queries are made, to make it of size exactly $16 \log n$. The parallel time remains $O(C(n)/\log n)$ while each round now is of size exactly $16 \log n$. This also implies that the total complexity is $c' = O(C(n))$.

At this point we can simulate $\mathcal{T}'$ exactly as is done in Theorem 2, using c' helpers. Each round of $\mathcal{T}$ is of $16 \log n$ queries, hence, the total overhead incurred by the faithful distribution of answers is linear in c'. Finally, to simulate the c' helpers by the original n processors, as in the proof of Theorem 2, the initial '0'-phase should be repeated for c'/n times as each group of c'/n helpers is simulated by a single processor and must receive c'/n independent copies of each variable. This contributes a total of $(c'/n) \cdot n = c'$ broadcasts in the first c'/n rounds. From there on, each processor simulates its helpers which is done in $O(c')$ total number of broadcasts. This concludes the proof. $\blacksquare$

5 Fault Tolerant Decision Trees and Protocols for other functions

We now describe several results for efficient fault tolerant decision trees for several families of Boolean functions. All the decision trees that we describe are deterministic and with parallel time $O(n/\log n)$. Hence, by Theorem 4, we obtain corresponding fault tolerant broadcast protocols.

We first note that fault tolerant decision trees can be composed to form a fault tolerant decision tree for the composition of functions.

Definition 1 *Let $0 \leq \epsilon < 1/2$. A decision tree is said to be an ϵ-fault tolerant decision tree for a function f if under the assumption that each query is erroneous with probability bounded by ϵ the output error is also bounded by ϵ.*

Let $f : \{0,1\}^n \longrightarrow \{0,1\}$ be computed by an ϵ-fault tolerant decision tree of complexity C_f and parallel time T_f, let $g_1, ..., g_n : \{0,1\}^m \longrightarrow \{0,1\}$ be Boolean functions each having an ϵ-fault tolerant decision tree of complexity C_i and parallel time T_i, $i = 1, ..., n$, respectively. Then the composition $f(g_1, ..., g_n) : \{0,1\}^{n \cdot m} \longrightarrow \{0,1\}$ can be computed by an ϵ-fault tolerant decision tree of complexity $C_f \cdot max_{i=1}^n C_i$ and parallel time $T_f \cdot max_{i=1}^n T_i$. This is due to the fact that one can 'compose' the algorithm for f with those for the $g's$ in the straight forward way (as the assumption on the input error bound is equal to the assumption on the output error bound). Namely, the algorithm for $f(y_1, ..., y_n)$ is applied and every time there is a query to y_i, a new run of the algorithm to g_i is made.

As an example let $[n]$ be partitioned into $m = \sqrt{n}$ disjoint 'blocks' $B_1, ..., B_m$, each of size m. We index n Boolean variables by $[m] \times [m]$ and let $f = \wedge_{j=1}^m \vee_{i \in B_j} x_{i,j}$. To compute f by a ϵ-fault tolerant decision tree we compute the top AND in $O(m)$ queries using Theorem 2 amplified so that its output error is bounded by ϵ. Each query to a sub function in the AND is answered by an independent run of a decision tree of an OR$_m$ function.

The above can be generalized to any AC_0 formula by replacing $C_f \cdot max_{i=1}^n C_i$ above with $\sum_{i=1}^m C_i$ in the case that f is the OR function. A Boolean formula here is just a Boolean circuit with $\wedge, \vee$ as gates with unbounded fan-in, and fan-out=1. The inputs to the circuit are literals, namely Boolean variables or their negation. The total size of the formula is the number of literals and its depth is the longest path from an input to the output gate, measured in term of the number of gates. A formula is said to be an $AC_0(s,d)$ formula[2] if it has depth d and size s. Thus the example shown above is of an $AC_0(n,2)$ formula. The class AC_0 is the standard notation for the union of all $AC_0(poly(n), d)$ formulae where $d = O(1)$ and n is the number of variables of the function.

Theorem 5 *There are universal constant $\alpha, \beta > 1$ such that if $f : \{0,1\}^n \longrightarrow \{0,1\}$ can be computed by an $AC_0(s,d)$ formula, then f has a fault tolerant decision tree of complexity $O(\alpha^d \cdot s)$ and $O(\beta^d \log^d n)$ parallel time.*

In order to prove the theorem we will need to generalize our solution for the OR function to the case in which each query has its own cost.

[2]This is slightly non-standard: AC_0 is usually defined for circuits and with size s that is polynomial in n. Then, the definition for formulae and circuits coincide. Here we have defined it for formulae and general size s, although the application will only make sense for polynomial s (in fact linear or just slightly above linear). We also note that usually $AC_0(s,d)$ denote the class of functions/languages that can be computed by $AC_0(s,d)$ circuits rather than the class of circuits itself.

5.1 Complexity of decision trees with non-uniform queries costs

So far we have always considered queries of unit cost. It is of interest to consider non-uniform costs. That is, queries to some variables cost more than to others. This is used here for the proof of Theorem 5 and may be interesting on its own.

Definition 2 *Let $c : [n] \longrightarrow \mathbb{N}$ be a cost function (identified with a cost function on the variables $\{x_1, \ldots, x_n\}$). A decision tree for $f : \{0,1\}^n \longrightarrow \{0,1\}$ relative to the cost c is of complexity w, if for every possible root-to-leaf path the sum of the costs of the variables that are queried is at most w.*

Obviously, this definition carries on to deterministic trees, randomized trees (and expected cost) and fault tolerant trees. With this definition, a standard decision tree is just an instance of the priced case with unit costs. The error of a decision tree in the fault tolerant model remains the same, only its complexity changes. The parallel time also remains the same (and for the optimal tree it could depend on the cost function). The following theorem generalizes Theorem 1.

Theorem 6 *Let $c : [n] \longrightarrow \mathbb{N}$ be a cost function. Then there is a fault tolerant decision tree for $\mathrm{OR}(x_1, ..., x_n)$ with respect to the cost c, of complexity $O(\Sigma_i c(i))$ and parallel time $O(\log n)$.*

Proof: Let $C = \Sigma_i c(i)$ and let $k = \lceil \log C \rceil$. For a subset $S \subseteq [n]$ we denote $c(S) = \Sigma_{i \in S} c(i)$ and $\mathrm{OR}(S) = \vee_{i \in S} x_i$.

We may assume w.l.o.g that all costs are powers of 2, otherwise we replace each cost by the smallest power of 2 that is greater or equal to it. This increases the total cost by at most a factor of two. We assume w.l.o.g that the error of any single query is bounded by $\epsilon \leq 1/100$. We will show that for any $S \subset [n]$, $\mathrm{OR}(S)$ can be computed in complexity $O(c(S))$ and parallel time $O(\log |S|)$. We first consider a special case where all costs are *distinct* powers of 2.

Case 1, distinct costs: Let $S \subseteq [n]$ be a subset of the variables that contains no two variables of the same cost. The following is an algorithm that computes $\mathrm{OR}(S)$ with complexity $O(c(S))$, parallel time $O(1)$ and error bounded by $1/100$.

Algorithm $A_1(S)$: here S is a set of variables whose costs are distinct powers of 2.

Let $\{y_1, \ldots, y_r\}$ be a re-enumeration of the variables in S so that $c(i) > c(i+1)$.

For $i = 1, \ldots, r$, we query y_i $100i + 1$ times and set z_i to be the majority of the answers. We then output '0' for $\mathrm{OR}(S)$ if for every i $z_i = 0$ and 1 otherwise.

Claim 5.1 *Algorithm $A_1(S)$ computes $\mathrm{OR}(S)$ with respect to the cost function c with error bounded by $1/100$, complexity $O(c(S))$ and parallel time 1.*

Proof: Obviously the error of the algorithm $A_1(S)$ is bounded by the event that for some i, $z_i \neq y_i$. By Chernoff bound [1], $\mathrm{Prob}(z_i \neq y_i) < 101^{-i}$ for every fixed i. Hence

Prob$(\exists i,\ z_i \neq y_i) \leq \sum_i 101^{-i} \leq 1/100$. The complexity of the algorithm is $\Sigma_i(100i+1)c(i)$. Recall that by assumption, $c(i),\ i = 1,\ldots,r$ are distinct and each is a power of 2. Let $c(i) = 2^{k_i}$, then the expression for the complexity becomes $\sum_{i=1}^{r}(100i+1)2^{k_i} \leq \Sigma_{i=1}^{r}(100i+1)2^{k_1-i+1} = O(2^{k_1}) = O(c(S))$. The parallel time is obviously just 1. ∎

We now construct an algorithm for the general case. The algorithm will be recursive. It will follow the general ideas of the algorithm for uniform costs (the algorithm of Observation 2.2) and will use at times the algorithm A_1 of Case 1 above.

Recall that the costs are assumed to be powers of 2. Hence, there are n_i variables, each of cost 2^i for $i = 0, 1, \ldots, k$. Formally we denote by $Alg(T)$ the algorithm, where $T \subseteq [n]$ is a subset of the variables on which the OR has to be computed. To compute OR$(x_1, \ldots, x_n)$ Alg will be called with $T = [n]$. The cost c is fixed and assumed to be concentrated only on powers of 2 as explained above.

For every subset $T \subseteq [n]$ we fix a partition of T, $T = S_1 \cup S_2$ where $S_1 \cap S_2 = \emptyset$, S_1 contains $n'_i \equiv 0(mod\ 4)$ variables of cost 2^i for each $i = 0, 1, \ldots, k$, and S_2 contains at most 3 variables of the same cost. Obviously such a partition is always possible to find by grouping together variables with the same costs into blocks of size 4 and putting them into S_1 while putting the remaining variables (at most 3 of each cost) into S_2.

Following is the formal description of Algorithm $Alg(T)$.

$Alg(T)$**:**

computing OR(S_2)**:** We partition S_2 into at most three sets S_2^1, S_2^2, S_2^3 each with variables of distinct costs. Then we run $A_1(S_2^i),\ \ i = 1, 2, 3$ and take the OR of the three results.

computing OR(S_1)**:** Recall that the number of variables that have a certain fix cost in S_1 is divisible by 4. We split the variables of each cost into blocks of size 4, query all the variables once and chose from each block the variable with the smallest index in the block for which the value 1 is obtained (if there is no such variable, we choose the one with the minimal index in the block). Let T_1 be the resulting subset of variables.

We next call $Alg(T_1)$ to compute recursively OR(T_1), for 3 independent times and take the majority of the 3 results as the output for OR(S_1).

We note that in the base case the algorithm always stops the recursion when $S_1 = \emptyset$.

combining the partial results: The final output is just OR$(S_2) \vee$ OR(S_1).

We note that for the case of uniform costs, the algorithm essentially reduces back to that of Observation 2.2.

We first analyze the error of the algorithm. By Claim 5.1 OR$(S_2^i),\ \ i = 1, 2, 3$ is computed with error bounded by 1/100, complexity $O(c(S_2^i))$ and parallel time 1. Thus OR(S_2) is computed with total error bounded by 3/100, complexity $O(c(S_2))$ and parallel time 1. (Note that the hidden constant in the expression for the complexity is a universal constant - it does not depend on n or c, neither it depends on ϵ as long as $\epsilon \leq 1/100$).

Let us assume inductively on the size of $|S_1|$ that the error of $Alg(T)$ is at most $\delta = 1/10$. For the base case $S_1 = \emptyset$ hence the error is determined by the computation of OR(S_2), which is indeed lower than 1/10, as shown above.

We now assume that this was already proved for any T' for which $|S_1| < 4\ell$. Let $T \subseteq [n]$ be such that in the partition as defined in the algorithm $|S_1| = 4\ell$ (note that by assumption $|S_1| \equiv 0(\mod 4)$), and consider $Alg(T)$. Then, an error in computing $\text{OR}(S_1)$ occurs either if the variable with the minimum index that is '1' does not get to be included in T_1, or if there is an error in one of the three recursive calls to $Alg(T_1)$. The first event happens with probability $4\epsilon \leq 4/100$, since for this to happen at least one on the four queries to the variables in the block containing the minimum indexed variable that is '1' has to be answered erroneously. The second event happens if at least two of the three independent recursive calls return an erroneous answer. By induction this is bounded by $3\delta^2 \leq 3/100$. Thus the total error in computing $\text{OR}(T) = \text{OR}(S_2) \vee \text{OR}(S_1)$ is at most $\frac{3}{100} + \frac{4}{100} + \frac{3}{100} = 1/10$ as required.

To bound the total complexity, let $f(B)$ denote the total complexity of $Alg(T)$ on a subset of variables whose total cost $c(T) = B$. We get the recurrence $f(B) \leq O(B) + 3f(B/4)$. This is true as in S_1 each block contains 4 variables of the *same* cost and hence the remaining set T_1 on which Alg is called recursively has total cost $c(T_1) = c(S_1)/4 \leq B/4$. Obviously, this implies that $f(B) = O(B)$.

To assess the parallel time we note that each recursion level is done in one round. Hence, the parallel time is bounded by the recursion depth which is $O(\log |T|)$. This is true as in each recursive call the size of set of inputs is decreased by a factor of at least 4. This concludes the proof of Theorem 6. ∎

We now are ready to prove Theorem 5.

Proof: (of Theorem 5). Fix any $\epsilon < 1/100$. We use induction on the structure of the formula for f. Assume w.l.o.g that $f = \vee_{j=1}^{m} g_i$ where g_i has $AC_0(s_i, d-1)$ formula, namely, of depth $d-1$ and size s_i. Let α and β be the hidden constants in the expressions for the cost and parallel time, respectively, for computing the OR relative to a cost function in Theorem 6. By induction every g_i has a $\frac{1}{100}$-fault tolerant decision tree $\mathcal{T}_i$ of complexity bounded by $\alpha^{d-1} \cdot s_i$ and parallel time $O(\beta^{d-1} \log^{d-1} n)$. Then the following is an $\frac{1}{100}$-fault tolerant decision tree for f: Apply an $\frac{1}{100}$-fault tolerant decision tree for $\text{OR}(y_1, ..., y_m)$ with costs $c(i) = \alpha^{d-1} s_i, i = 1, \ldots, m$ where each query to a variable y_i is replaced by an independent run of $\mathcal{T}_i$. Obviously the resulting decision tree is a $\frac{1}{100}$-fault tolerant decision tree for f. Now, the existence of a $\frac{1}{100}$-fault tolerant decision tree for $\text{OR}(y_1, ..., y_m)$ of complexity bounded by $\alpha \cdot \Sigma c(i)$ and parallel time $\beta \log m$ is asserted by Theorem 6 (using amplification to reduce the output error below $\frac{1}{100}$) and the induction hypothesis. Hence the total complexity and parallel time is as claimed. ∎

Corollary 5.1 *If f can be computed by an $AC_0(O(n), d)$ formula for some $d = O(1)$ then f has a fault tolerant decision tree of complexity $O(n)$ and parallel time $O(\log^d n)$.*

Another class of functions that can be computed efficiently in the fault tolerant decision-tree model is the class of functions that have $O(1)$ size 1-witness[3] or $O(1)$ size 0-witness. King and Kenyon, [15], proved that any function of $O(r)$ size 1-witness (not necessarily

[3]a 1-witness is a minimal assignment to a subset of the variables that ensures that the function is 1. A 0-witness is defined analogously. For exact definitions and more material on this see [27].

monotone) can be computed by a fault tolerant $O(n \log r)$ size decision tree. However, their algorithm may have linear parallel time. We show;

Theorem 7 *Let f be a Boolean function whose 1-witness size is at most $r = O(1)$, then f can be computed by a fault tolerant decision tree of complexity $O(n)$ and parallel time $O(\log^2 n)$.*

The dependence of our algorithm on r, the witness size, is given explicitly in Theorem 8 and is much worse than that of [15]. As we aim for linear algorithms (that is for $r = O(1)$) we do not try to optimize this dependence. The algorithm is quite different from that of [15]. The proof will follow from Theorem 8 for monotone functions.

As a corollary of Theorem 7 and Theorem 4 we get

Corollary 5.2 *Let f be a Boolean function f whose 1-witness size or whose 0-witness size is at most $r = O(1)$. Then f can be computed by an $O(n)$ fault tolerant broadcast protocol.* $\blacksquare$

A monotone function has *minterm-size* $\leq r$ if there are subsets $S_1, \ldots, S_t \subseteq [n]$ each of size at most r, called *minterms*, and such that f can be written as $f = \vee_{i=1}^{t} \wedge_{j \in S_i} x_j$. Thus, The OR has minterms of size 1 and the Majority function on n variables has minterms of size $\lceil \frac{n+1}{2} \rceil$.

Theorem 8 *Let f be a monotone Boolean function whose minterm size is at most r, then f can be computed by a fault tolerant decision tree of complexity $O(nr \log^2(2r))$ and parallel time $O(r^2 \log^2 n)$.*

Proof of Theorem 8: As we apply the result for $r = O(1)$ we do not try to optimize the dependence of the complexity in r. In addition, we omit $\lceil\ \rceil$ signs even when the numbers must be integers to improve readability.

We assume in the following that $r < \log n$; for $r \geq \log n$ the result is trivial as any function has a fault tolerant decision tree of complexity $O(n \log n)$ and parallel time $O(1)$.

We will need the following simple observation.

Observation 5.1 *Let $S \subseteq [n]$ be a set of variables and let k be an integer. There is a fault tolerant decision tree whose complexity is $O(|S|(\log k + \log |S|))$ and parallel time 1, that evaluates correctly all the variables in S with error probability at most $1/(20k^2)$.*

Proof: We query each variable $a \cdot (\log |S| + \log k)$ times and take the majority value of the answers as its 'computed value'. The constant $a = a(\epsilon)$ is chosen so that the probability that the majority value is incorrect is bounded by $1/(20|S| \cdot k^2)$. By Chernoff bound [1], there is indeed such constant a for any $\epsilon < 1/2$. Thus, the probability that any of the variables is evaluated wrongly is bounded by $1/(20k^2)$ $\blacksquare$.

Definition 3 *Let $S \subseteq [n]$ be a set of variables, the fault tolerant decision tree that is implied by Observation 5.1 will be denoted as* trivially k-evaluating S. *If there is a function $f : \{0,1\}^S \longrightarrow \{0,1\}$, that is, on variables indexed by S, then evaluating f by trivially k-evaluating S will be denoted as* trivially k-evaluating f.

Let $f : \{0,1\}^n \longrightarrow \{0,1\}$ be a monotone function on n variables whose minterm size is bounded by r. Let MIN(f) be the set of minterms of f. Thus $f = \vee_{P \in MIN(f)} \wedge_{j \in P} x_j$. We represent the collection of minterms of f of size exactly r by a hypergraph $H_f = ([n], E)$ in which every hyper-edge $e \in E$ is a minterm. A matching in H_f is a collection of pairwise disjoint edges. A cover of H_f is a subset $C \subseteq [n]$ such that for each $e \in E$ there is an $i \in C$ such that $i \in e$. Note that if M is a maximal matching in H_f then $\cup_{e \in M} e$ is cover of size $r \cdot |M|$. For a minterm $P \in H_f$ we denote by $P(x)$ the Boolean function $P(x) = \wedge_{i \in P} x_i$. Namely, $P(x) = 1$ whenever the minterm P is 1.

For $r = 1$ f is the OR function. We thus assume in the following that $2 \leq r < \log n$. We also assume in the following that the bound on the query error $\epsilon \leq 1/(20r^3 + 1)$ and explain at the end how this assumption affects the complexity. We note that although the algorithm is recursive (we use induction on r), this bound on the query error is fixed and does not changes during different calls for the algorithm when r decreases.

The top level idea is the following dichotomy into two cases:

case 1: If there is a cover C of H_f of size at most $(n/\log n)$ then we trivially n-evaluate the variables in C and restrict ourselves to the subfunction $f_1 = f|_{\bar{C}}$ that is defined on the variables outside C, by substituting in the evaluation for the variables in C. As each minterm is covered by C, it follows that f_1 has minterms of size bounded by $r-1$. Applying the induction hypothesis ends the proof in this case.

Case 2: If there is no cover of size at most $(n/logn)$ then by the observation above, every matching in H_f is of size at least $n/(r \log n)$. Let M be a matching in H_f of size $m = n/(r \log n)$. We may always assume the existence of such a matching as we can take a partial matching if the size of a certain matching is larger than the threshold above. Then, f can be written as $f_M \vee f'$ where $f_M(x) = \vee_{P \in M}(P(x))$ and f' is the function whose minterms are in $MIN(f) - M$. Note that f_M is an OR of m disjoint minterms.

In this case we first evaluate f_M using the ϵ-fault tolerant algorithm for OR (of Observation 2.2). If f_M evaluates to 1 then this implies that f is 1 too. Otherwise, if f_M evaluates to 0, assuming that this is the correct value of f_M, it implies that every minterm in M must contain a variable that is 0. Thus there exists a set of m variables each of value 0. We will show how to find a subset of such a set of size at least $m/2$ and restrict f to the rest of the variables. Thus our gain will be that either case 1 occurs, in which the restricted function has minterms of size $r-1$, or that case 2 occurs in which we end up computing a function on only $n - m/2$ variables.

We now formally present the algorithm as a recursive algorithm.

Algorithm A(f,n,r)
f is a monotone function on at most n variables and has minterms of size at most r.

For each variable x_i we keep a count $count^{(r)}(x_i)$ that equals to the number of 1 answers minus the number of 0 answers in all the queries to x_i.

1. If $r = 1$ then $f = \vee_{i=1}^{n} x_i$. Compute f by the algorithm in Observation 2.2.
2. If f depends on at most $n/logn$ variables we trivially n-evaluate f.
3. If there is a cover C of H_f of size at most $n/\log n$ then we trivially n-evaluate all variables in C. Now we may restrict ourselves to the subfunction $f_1 = f|_{\bar{C}}$ that is defined on those variables outside C, by substituting the evaluation for the variables in C. If f_1 is already determined then we are done. Otherwise, as each minterm in H_f is covered by C, it follows that f_1 has minterms of size bounded by $r - 1$. We compute f_1 by calling recursively $A(f_1, n, r - 1)$.
4. If there is no cover of size at most $n/logn$ then by the discussion above, every matching in H_f is of size at least $n/(r \log n)$. Let M be a matching in H_f of size $m = n/(r \log n)$ and let $f_M(x) = \vee_{P \in M}(P(x))$.
   (a) We compute f_M by applying the algorithm of Observation 2.2 for $\mathrm{OR}(y_1, \ldots, y_m)$, where we take $y_i = P_i(x)$. Every time we need to query y_i we start a new trivial r-evaluation of $P_i(x)$.

       If $f_M = \mathrm{OR}(y_1, \ldots, y_m)$ evaluates to 1, recall that the algorithm of Observation 2.2 also provides an index i for which $y_i = 1$. We trivially n-evaluate P_i where P_i is the minterm associated with y_i. If P_i evaluates to 1 then we stop and output 1 for the whole function.

   (b) Set $S = \{i | \; count^{(r)}(x_i) < 0\}$. If $|S| < m/2$ we stop and output the value 1 for f. The reason for this is that if $f_M(x) = 0$ it will be shown that $|S| < m/2$ with very small probability. Thus we don't really expect the algorithm to enter this case. However, this will allow us to guarantee that in the alternative, that is in step 4c below, n' is small enough.

   (c) Otherwise, if $|S| \geq m/2$ we substitute 0 for every variable in S and compute the restricted function, $f|_{\bar{S}}$, on $n' = n - |S|$ variables by calling recursively $A(f|_{\bar{S}}, n', r)$.

We first analyze the complexity: In steps 1, 2 a work of $O(n)$ is done. These serve as the base cases for the induction.

If step 3 occurs then the trivial evaluation of C takes $O(n)$ queries, after which we reduce the computation to a function whose minterms are of size at most $r - 1$. Otherwise, each trivial evaluation of $P(x)$ in step 4a takes $O(r \log r)$ queries. Computing $OR(y_1, \ldots, y_m)$ takes $O(m)$ such trivial evaluations using the algorithm for the OR. Thus this takes a total of $O(m \cdot r \log r) = O(n \log r / \log n)$ queries. To this we possibly need to add $O(r \log n)$ queries for the trivial n-evaluation of P_i in case that f_M evaluates to 1. After step 4a we either stop (at step 4b), or make a recursion call at step 4c.

Let $f(n,r)$ be the maximum number of queries of $A(f,n,r)$ for the worst case of f,n and r. We get the recurrence:

$$f(n,r) \leq \max\left\{O(n),\ O(n)+f(n,r-1), O(\frac{n\log r}{\log n}+r\log n),\ O(\frac{n\log r}{\log n}+r\log n)+f(n-\frac{m}{2},r))\right\}$$

The first $O(n)$ term in the max is for the base case (step 1 or 2). The second term is when step 3 occurs, the third is when we stop after step 4a or after step 4b, and the fourth term is when we end up doing step 4c.

The right hand side of the bound for $f(n,r)$ is dominated by the second and fourth terms in max{} (for our choice of r and m). As can be checked by substitution, $f(n,r) = O(nr\log(2r))$ solves the recursion. Recall that we assume a bound of $1/(20r^3+1)$ on the query error. This is achieved by simulating each query using $\Theta(\log r)$ real queries, which brings in another factor of $O(\log r)$, implying the claimed complexity.

The parallel time is bounded by $O(n/m)\cdot O(\log n)\cdot r = O((r\log n)^2)$. This is so as to reduce r to $r-1$ it either takes one round if step 3 occurs, or step 4 can be iterated $O(n/m)$ times and each path through step 4 is done in $O(\log n)$ rounds (spent on computing $\mathrm{OR}(y_1,\cdots,y_m)$).

We now analyze the error probability of the whole algorithm. Before we proceed we note that for a 1 valued variable x_i, the probability that there is a step in the algorithm (in the top recursion level) for which $count^{(r)}(x_i) < 0$ is upper-bounded by $\frac{p}{1-p}$ where p is the query error. This is by standard analysis in which the value the random variable $count^{(r)}(x_i)$ assumes is viewed as a random walk on the integers, see e.g [12] Chapt. 3. Plugging in $p < 1/(20r^3+1)$, we get that $count^{(r)}(x_i) < 0$ for a 1-valued variable x_i with probability at most $1/(20r^3)$. Similarly, a 0-valued variable becomes of positive count with the same probability.

Consider first a 1-input, x. We fix an arbitrary enumeration of the minterms of f and let P^* be the first minterm according to this enumeration for which $P^*(x) = 1$. The algorithm may result in an erroneous 0 answer in the following three cases: The first is if at step 2 or 3 there is an error in the trivial n-evaluation of C. By definition, each of these happens with probability at most $1/(20n^2)$. Step 2 may be entered only once during the whole algorithm. Step 3 may be entered at most $r-1$ times, as after each path through it r decreases by at least 1. Thus this contributes at most $r/(20n^2)$ to the total error. The second case is when at least one variable x_i belonging to P^* becomes of negative count and thus f may be restricted to be the 0-function at step 4c. There are at most r variables in P^* and each may become of negative count with probability at most $1/(20r^3)$, thus the probability that this event occurs during the whole algorithm is at most $r\cdot\frac{1}{20r^3} = 1/(20r^2)$. The only other case is an error at the base case for $r=1$, which by Observation 2.2 may be assumed to be say, 1/8. Thus the total error for x is at most $\frac{r}{20n^2}+\frac{1}{20r^2}+\frac{1}{8} \leq 1/3$.

To bound the error for a 0-input, $y \in f^{-1}(1)$, note that the only way the algorithm may end with a faulty 1 answer, in the top recursion level, is in the following three cases. The first is when the algorithm makes an incorrect evaluation of the variables in C at step 3. The second is when the function $f_M(y)$ evaluates to 1 followed by a faulty trivial n-evaluation of $P_i = 1$. Obviously each of this events happens with probability at most $1/(20n^2)$. The third case is if at step 4b less than $m/2$ of the variable became of negative count resulting in a 1-output. Note that if $f(y) = 0$ then also $f_M(y) = 0$. Recall also that f_M is the OR of m disjoint minterms. Thus if $f_M = 0$ then each of the m disjoint minterms in M must contain a variable that is 0. The probability that such a variable becomes of positive count is, as noted above, at most $1/(20r^3)$. Moreover, for different variables these are independent events. Thus Chernoff bound [1], asserts that less than $m/2$ variables become of negative count with probability less than $exp(-\Omega(m)) \leq 1/(20n^2)$.

Thus, an error for y occurs in the top recursion level with probability at most $3/(20n^2)$. There are at most $2r \cdot (n/m)$ recursion calls, since step 4c may be entered at most $2n/m$ times after which r decreases by at least one. This implies that $f(y)$ is computed erroneously with probability at most $\frac{6r^2 \log n}{20n^2} \leq 1/3$. ■

Proof of Theorem 7: Let f be a Boolean function on the variables $x_1, \ldots, x_n$, with r-size 1-witnesses. Then f can be written as $f = \vee_i W_i(x)$ where each W_i corresponds to a distinct 1-witness and is a conjunction of at most r-literals (see e.g [27] for more details on this). Let $z_1, \ldots, z_n$ be n new variables. We replace every appearance of $\bar{x_i}$ (the negation of x_i) in the formula for f, with z_i. This results in a new monotone function f' on $2n$ variables with minterm size at most $2r$, and for which f is a subfunction. We then evaluate f' on the new variables while treating x_i, z_i as independent variables. The Theorem now follows from the monotone case. ■

6 Recovering the whole input word

As mentioned in Section 1, Gallager's protocol is a fault-tolerant algorithm that computes the whole input word in $O(n \log\log n)$ broadcasts. Theorem 3 implies that the whole input word can be computed in $O(n \cdot r \log r)$ broadcasts, provided that there are at most r 1's in it (just by finding the OR r times, each with error probability reduced to $1/(3r)$). Can we do better? We get.

Theorem 9 *For any r, there is a fault tolerant protocol of complexity $O(n \log r)$ that for every input word, it either computes the whole word, or it finds at least r positions in the word that are '1'.*

Proof: Note that for r such that $\log r = \Omega(\log\log n)$ the whole word can be computed in $O(n \log\log n)$ by Gallager's result. Hence, we assume in the following that $\log r = o(\log\log n)$.

Instead of describing a protocol for the broadcast model, we describe an algorithm for the noisy decision tree model that ends by finding r 1's (if exist). This algorithm can then be turned into a protocol for the broadcast model as asserted by Theorem 4.

The decision tree algorithm is as follows. We assume that the query error is bounded by $1/(10r^3)$. This is done using simple amplification at the cost of $\Theta(\log r)$ queries of 1/3-error bound. We call such query a superquery. We partition $[n]$ into $m = r^2$ disjoint blocks, each of size n/m. The algorithm first computes the OR in the first block using the algorithm in Observation 2.2. Recall that assuming a query-error bounded by ϵ the algorithm in Observation 2.2 will end by outputting the first variable in the block that is 1 (if such exists) with probability $1-\epsilon$. Once such a variable x_{i_1} of value 1 is obtained then i_1 is taken to be in the set of indices that the algorithm will return. In this case, the algorithm will remove i_1 from the block and repeat its action. If the value of the OR on the block is 0, then the algorithm moves to the next block. It does so until it either collects r indices on which it received an answer 1, or the OR in every block evaluates to 0.

The final output is the set of indices thus obtained. In each round of computation $O(n/m)$ superqueries are made in order to compute the OR of the variables in a block. Thus in total, at most $\max\{r,m\} \cdot O((n/m)\cdot) = O(n)$ superqueries are made which is equivalent to $O(n \log r)$ simple queries. The parallel time is $\max\{r,m\} \cdot \log n = O(\log^2 n)$ and the error probability for any phase out of the $\max\{r,m\}$ phases is bounded by $1/(10r^3)$ thus in total it is at most $1/(10r) \leq 1/10$. ■

7 Algorithms for OR of $\log^* n$ parallel time

We first describe here a fault-tolerant decision tree algorithm for the OR that has simultaneous $O(n)$ total complexity and $O(\log^* n)$ rounds. We then indicate how to simulate it in the broadcast model. Recently [14] have proved that this is best possible for the decision tree model. For the broadcast model we still can't rule out a protocol of total complexity $O(n)$ and $O(1)$ rounds.

Theorem 10 *There is a fault-tolerant decision tree algorithm that computes the* OR *of n variables, using $O(n)$ queries in $O(\log^* n)$ rounds.*

Proof: In the following we assume that query error bound is $\epsilon < 1/6$. In particular, by Chernoff bound (e.g [1]), if we query a variable m times and take the majority answer then the probability that the value is wrong is at most 2^{-m}.

We adopt the strategy of the algorithm of Observation 2.2, but unfold the recursion using a somewhat different trade-offs between number of queries and error probability. We will have at each round $i = 1, \dots$ a set $I_i \subseteq [n]$ of variables that is supposed to hold the minimum index $\ell \in [n]$ for which $x_\ell = 1$ (if such ℓ exists). Then, at stage i, every variable $x_j, j \in I_i$ is going to be queried k_i times, and a majority vote, M_j, on those values is going to represent its value for the ith round. Based on $M_j, j \in I_i,$ the next subset of candidates $I_{i+1} \subseteq I_i$ is going to be formed as follows: we are going to divide I_i into blocks of size b_i. Each block will contribute a single variable to I_{i+1} - namely the variable of minimum index in the block that has value '1' (or an arbitrary variable in the block if all evaluated to '0'). This is done until I_i is narrowed down to less than $n/\log n$ at which point, every variable can be queried $100 \log n$ times, enough to know its value with confidence better than $1 - 1/n^2$.

Formally, let $n_i = |I_i|$. We denote by k_i the number of queries that each variable is going to be queried at round i, and by b_i the block size at round i. Querying each variable k_i times and taking majority will implies an error $e_i \leq 2^{-k_i}$ on the value we take for such a variable at round i. We then divide I_i into blocks of size $b_i = 2^{k_i} \cdot 2^{-(i+2)}$, out of which the variable of minimal index of value 1 will enter I_{i+1}.

We start with round 1 in which $I_1 = [n]$ thus $n_1 = n$, and with $k_1 = 100$ and $b_1 = 2^{k_1 - 3}$. For general $i \geq 2$ we set $n_i = n_{i-1}/b_{i-1}$, $k_i = \frac{100n}{n_i} \cdot 2^{-(i-1)}$ and $b_i = 2^{k_i} \cdot 2^{-(i+2)}$. Thus assuming that the size of I_{i-1} is n_{i-1}, and that we divide I_{i-1} into blocks of size b_{i-1} from which one member is to enter I_i then the size of I_i is exactly n_i.

To analyze the total complexity, recall that at round i we query k_i times each variable in I_i. Thus, a total of $n_i \cdot k_i = 100n2^{-i+1}$ queries are made at round i. The sum over all rounds is $O(n)$.

To analyze the number of rounds, note that $\frac{k_{i+1}}{k_i} = 0.5\frac{n_i}{n_{i+1}} = 0.5b_i = 0.5 \cdot 2^{k_i} \cdot 2^{-(i+2)}$. In other words, $k_{i+1} = k_i \cdot 2^{k_i} \cdot 2^{-i-3} \geq 2^{k_i}$. Thus after $\log^* n$ rounds $k_i = \Omega(n)$, hence $n_i = O(1)$.

Finally, to analyze the error probability, note that if all variables are 0 then there is no error before the last phase. Assume then that ℓ is smallest for which $x_\ell = 1$. Then the algorithm errs before the last phase if $\ell \notin I_i$ for some i. However, by our setting, the error in the value that is computed for each queried variable at round i is $e_i = 2^{-k_i}$ while $b_i = 2^{-(i+2)} \cdot \frac{1}{e_i}$. Hence, conditioned on the event that $\ell \in I_i$, the probability that there is an index in the block containing x_ℓ (that is, among the b_i variables) that is evaluated erroneously is at most $2^{-(i+2)}$. Thus, conditioned on the event that $\ell \in I_i$, the probability that $\ell \notin I_{i+1}$ is at most $2^{-(i+2)}$. Summing up for every i we conclude that the probability that ℓ is in every I_i, $i = 1, \ldots$ is at least 3/4. ■

Theorem 11 *There is a 1/4-fault tolerant protocol, in the noisy broadcast model, that for any input $x \in \{0,1\}^n$ it ends with all processors knowing the index ℓ which is the smallest for which $x_\ell = 1$ or arbitrary index if* $\mathrm{OR}(x) = 0$. *The protocol makes $O(\log^* n)$ rounds and total complexity $O(n)$.*

Proof: The top level idea is to simulate the algorithm of Theorem 10. The difficulty is that the faithful distribution of the winners in each round is inherently sequential due to the centralized control. To remedy this we 'distribute' the global knowledge. In the following description, a processor P_j sometimes needs to broadcast a 'fresh-value' of some variable x_i. We assume that it contains a sequence of independent copies $x_i^{(1)}, x_i^{(2)}, \ldots$ of x_i, each which could be erroneous with probability ϵ, and each time it needs a fresh copy it takes the next value in this sequence.

The algorithm will consist of $O(\log^* n)$ rounds. In each round $i = 1, \ldots$ there will be a set $I_i \subseteq [n]$ that should contain the minimum index ℓ such that $x_\ell = 1$. Similarly to the algorithm of Theorem 10, the size of I_i will decrease very rapidly which will provide the $O(\log^* n)$ bound on the number of rounds. Formally, in each round, $[n]$ is partitioned into disjoint blocks of uniform size. The partition $\mathcal{B}_{i-1}$ of round $i-1$ will be a refinement of the partition $\mathcal{B}_i$. That is, each block of $\mathcal{B}_i$ is a union of b_i blocks of the partition $\mathcal{B}_{i-1}$. Let s_i denote the size of the blocks of $\mathcal{B}_i$. If round $i-1$ is successful for a block $B \in \mathcal{B}_{i-1}$ then the smallest index $j \in B$ for which $x_j = 1$ is the 'winner' in B at round $i-1$ and it is known to every processor in B. Then in round i we do the following, in each block $B \in \mathcal{B}_i$:

Round i:

Recall that for such B, $B = \cup_{j=1}^{b_i} B_j$ where $B_j \in \mathcal{B}_{i-1}$. Formally we set $b_0 = 1$.

1. For every $j = 1, \ldots, b_i$, every processor among the first b_{i-1} processors in B_j broadcasts a 'fresh copy' of the value of w where w is the winner of B_j (at round $i-1$).

2. Every processor $P_k, k \in B$ hears for every $j = 1, \ldots, b_i$, the b_{i-1} broadcasts of every B_j-winner. He takes the majority of the b_{i-1} values per winner and takes the result as his own idea of the winner's value in each $B_j \in B$.

3. Based on the winner-values, each processor in B decides what is the smallest indexed block, B_j, from which a winner of value 1 is obtained, and takes this block as the block-winner of B.

4. At this point all the processors in B know the block B_j that contains x_r, the minimum indexed variable of value '1' in B. However, they don't know r itself. The purpose of this step is to make r known to every body.

   If $i = 1$ go to next step, for $i \geq 2$, assume that the winner is from a certain block B_j (every processor may have a different idea of who is the block-winner - we will show that with very high probability all processors have the same block-winner in mind). The first b_{i-1} processors in B_j broadcast the identity of the winner. This identity is fully specified by $\log s_{i-1}$ bits that are broadcast in one round - each bit being broadcast by $b_{i-1}/(\log s_{i-1})$ processors. In addition, to make this oblivious, in every other block $B_k, k \neq j$ the first b_{i-1} processors broadcast an arbitrary bit in this step.

5. Each processor $P \in B$ takes the majority of the values of the individual identity bits of the winner and set this as the identity of what he perceives as the unique winner for B.

The algorithm proceeds this way until round i in which $\mathcal{B}_i$ contains just one block. At this point there is only one winner known to all processors.

What needs to be defined next is how the whole thing is started, what is b_i and how the 'fresh-copies' are obtained.

We assume in the following that the query error is at most ϵ is such that $2\epsilon \leq 2^{-64}$. We start with round 1 in which $I_1 = [n]$ we formally set $\mathcal{B}_0 = [n]$; that is, each block is a singleton and $b_0 = 1$. We set $b_1 = 2^{12}$ and for $i \geq 2$ we set $b_i = 2^{b_{i-1}/(3 \log b_{i-1})}$. Recall that s_i is the block size of $\mathcal{B}_i$ and thus $s_i = b_i \cdot s_{i-1} = \prod_{j=1}^{i} b_j$. Before we proceed we note that:

Claim 7.1 *For every $i \geq 1$*

- $s_i = \prod_{j=1}^{i} b_j \geq 2^{i-2} b_i \cdot b_{i-1}$.
- $2^{b_i/(\log s_i)} \geq 2^{i+6} \cdot s_{i+1} \cdot \log s_i$.
- $2^{b_i} \geq s_{i+1}^2 \cdot 2^{i+6}$.
- *for $i = 2 \log^* n$, $b_i > n$.*

Proof: The proof of the first, second and fourth items is by induction on i. The third item is a weaker inequality than the second. ■

We first analyze the algorithm assuming that fresh-copies are available as needed.

The complexity of step 1 of round i is $\frac{n}{s_i} \cdot b_i \cdot b_{i-1} \leq n \cdot 2^{-i+2}$ (the last inequality is by the first item in Claim 7.1). The same complexity is incurred at step 4. The other steps do not involve any broadcasts. Thus the overall complexity is at most $8n$.

By the fourth item in Claim 7.1, after $i = 2\log^* n$ rounds the block size (which is at least b_i) is more than n and thus $|I_i| \leq 1$. Thus the number of rounds is at most $2\log^* n$.

We now bound the error probability. Let x be any input for which $\mathrm{OR}(x) = 1$ (there is nothing to prove if $\mathrm{OR}(x) = 0$), and let ℓ be the smallest for which $x_\ell = 1$. For $i = 1, \ldots$ let $B^{(i)} \in \mathcal{B}_i$ be the block that contains ℓ and G_i be the event that at round i, ℓ is known as the winner of the corresponding block $B^{(i)}$ by every processor in $B^{(i)}$. Also, set $B^{(0)} = \{x_\ell\}$ and $G_0 = True$ (namely the event of probability 1).

Claim 7.2 *Assuming the availability of 'fresh copies' as required by the algorithm at step 1 of every round, then* $\forall i, \quad Prob(G_i \mid G_{i-1}) \geq 1 - 2^{-(i+4)}$.

Proof: For $i = 1$ let $B = B^{(1)}$ be the block that contains ℓ. At step 1 of round 1 every processor in B knows each bit of B with error probability $2\epsilon \leq 2^{-64}$ (the factor 2 here is due to the fact that it is not the bit itself that is being broadcast, rather it is a 'fresh-copy' of it). Hence by the union bound, ℓ is known to be the winner for B by all processors in B with probability at least $1 - b_1^2 \cdot 2^{-64} = 1 - 2^{-40}$.

For general i, at step 1 of the round a fresh copy of x_ℓ is broadcast by b_{i-1} processors in $B^{(i-1)}$ (which by the conditioning know ℓ). Thus any processor which hears this and takes the majority of the results, knows that the value that is being broadcast by processors in $B^{(i-1)}$ is 1 with error probability at most $2^{-b_{i-1}}$. For any other block, $B \subseteq B^{(i)}$, $B \neq B^{(i-1)}$, that contains indices smaller than ℓ, the value of x_r for an arbitrary index $r \in B$ is being broadcast by each of the first b_{i-1} processors in B (this index might be different for different processors), however, since ℓ is the minimal for which $x_\ell = 1$ the true value of the corresponding broadcast by any processor in B is 0. Since the fresh copy that is actually being broadcast is correct with probability ϵ, the probability that any fixed processor in $B^{(i)}$ will hear a 1 from the block B is less than $2^{-b_{i-1}}$. Hence, by the union bound, the probability that there exists a processor in $B^{(i)}$ that does not know that the winner comes from $B^{(i-1)}$ is at most $b_i \cdot s_i 2^{-b_{i-1}} \leq 2^{-(i+5)}$, where the last inequality is by the third item in Claim 7.1.

We conclude that in the beginning of step 4 of round i, all processors in $B^{(i)}$ know that the winner of the round comes from $B^{(i-1)}$ with probability $1 - 2^{-(i+5)}$. Let us denote this event as D_i.

The identity of ℓ, that is the $\log s_{i-1}$ bits that are necessary to specify it inside $B^{(i-1)}$, are being broadcast each by $b_{i-1}/(\log s_{i-1})$ processors at step 4. Thus, after taking majority, the error per bit per receiver is at most $2^{-b_{i-1}/(\log s_{i-1})} \leq 2^{-(i+5)} \cdot (s_i \cdot \log s_{i-1})^{-1}$ (where the last inequality is by the second item in Claim 7.1). By the union bound, conditioning on D_i, every processor in $B^{(i)}$ knows the identity of ℓ (that is all the $\log s_{i-1}$ bits specifying it) correctly with error probability at most $2^{-(i+5)}$.

We conclude that $\text{Prob}(G_i | \, G_{i-1}) \geq 1 - 2^{-(i+4)}$. ■

Summing for all i, Claim 7.2 immediately implies that with probability 15/16 all G_i's occur, which is the success probability of the protocol.

It is left to explain how in each round, in each block, fresh-copies of the block winner are obtained for step 1 of the round. To do this note that at round i, b_{i-1} fresh copies are being used in every block B_j of round $i-1$. Thus altogether at most $8n$ fresh copies are needed (as sum is less than the total complexity). We thus use $8n$ helpers to supply those fresh copies. Each helper is designated to a certain block in a certain round and it will just broadcast once, one bit from its collection of fresh copies. We begin the protocol with round 0 in which each processor broadcasts its value. Thus each helper has one independent fresh copy of each variable. We will need to make sure that the b_{i-1} helpers that are designated to a block $B_j \in B$ know the winner of the block too. For each particular block this happens with the same probability that the processors that are members of the block know the correct unique winner. Thus the total error will just double and become at most 1/8. Finally, simulating the $8n$ helpers is done exactly as described in the protocol of Theorem 2. ■

We note that the protocol of Theorem 11 does not actually computes the OR, rather it ends with the smallest ℓ for which $x_\ell = 1$ (if such exists). This is somewhat similar to the situation in the protocol of Theorem 3 after the identity of ℓ is made known to everybody. To make the value of x_ℓ, which is the value of the OR, known to all processors, two final rounds are used. In the first every processor broadcasts its value and then every processor broadcasts the value it heard from P_ℓ. Finally, each processor takes the majority value it hears in the final round as the result of the computation.

This obviously implies:

Corollary 7.1 *There is a fault tolerant protocol that computes* OR_n *using* $O(\log^* n)$ *rounds and* $O(n)$ *total complexity.* ■

8 Further work and open problems

We have demonstrated that via fault tolerant decision tree algorithms we can construct fault tolerant broadcasts protocols for many functions, including symmetric and non-symmetric examples. The recent result of [13] solves the 'whole-bit' problem of Gallager. However, we still don't know the complete answer to Gallager's question, namely, whether there is any Boolean function that requires super linear number of broadcasts. The natural candidate for such function is the parity of all bits. It should be noted though, that [13] showed that the parity, as well as any symmetric function, can be computed in the corresponding statistical decision tree model in linear number of broadcasts.

We use here extensively a reduction to algorithms for the noisy decision tree model. It should be noted that the noisy decision tree model is 'much weaker' than the broadcast model; Feige et al. [7] proved a lower bound of $\Omega(n \log n)$ queries for the majority (and parity), while by Gallager those functions can be computed using $O(n \log \log n)$ broadcasts.

Concerning noisy decision trees; it would be nice to come up with other families of Boolean functions for which efficient (linear query complexity) algorithms exist, or ultimately give a characterization of such functions in terms of other natural parameters.

Finally, nearly everything about computing the OR function in both the noisy decision tree model and the broadcast model is known. There is still though one piece of information missing: Is it possible to compute the OR in $O(1)$ rounds and total complexity $O(n)$ in the broadcast model?

References

[1] N. Alon and J. H. Spencer, **The Probabilistic Method**, Second Edition, Wiley, New York, 2000.

[2] H. Buhrman, I. Newman H. Röhrig, R. de Wolf, *Robust Polynomials and Quantum Algorithms*, special issue of STACS05 in Journal of Theory of Computing Systems, 40(4), 379-395, 2007.

[3] R. L. Dobrushin and S. I. Ortyukov. *Upper bounds for the redundancy of self-correcting arrangements of unreliable functions*, Problems of Information Transmission, 13, 203-218, 1977.

[4] A. El Gamal, *Open problems presented at the 1984 workshop on Specific Problems in Communication and Computation* sponsored by Bell Communication Research.

[5] W. Evans and N. Pippenger. *Average-Case Lower Bounds for Noisy Boolean Decision Trees*, SIAM Journal on Computing 28(2), 433-446, 1999.

[6] U. Feige and J. Killian. *Finding OR in noisy broadcast network*, IPL 73, 69-75, 2000.

[7] U. Feige, P. Raghavan, D. Peleg, E. Upfal, *Computing with Noisy Information*, SIAM J. Comput. 23(5):1001-1018, 1994.

[8] P. Gács, *Reliable computation with cellular automata*, JCSS 32, 15-78, 1986.

[9] P. Gács and A. Gál, *Lower Bounds for the Complexity of Reliable Boolean Circuits with Noisy Gates*, IEEE Trans. Info. Theory, 40(2), (1999), 579-583.

[10] R. G. Gallager, *Finding parity in simple broadcast networks*, IEEE Trans. on Information Theory 34, 176-180, 1988.

[11] J. Justesen, *A class of constructive asymptotically good algebraic codes*, IEEE Transactions on Information, 18, 652-656, 1972.

[12] S. Karlin and H. M. Taylor, **A First course in stochastic processes**, Second Edition, Academic Press, 1975.

[13] N. Goyal, G. Kindler, M. E. Saks, *Lower Bounds for the Noisy Broadcast Problem*, Proceedings 46th IEEE Symp. Foundations of Computer Science (FOCS), 40-52, 2005.

[14] N. Goyal, M. E. Saks, *Rounds vs queries trade-off in noisy computation*, Proceedings 16th Annual ACM-SIAM Symp. Discrete Algorithms (SODA), 632-639, 2005.

[15] C. Kenyon and V. King, *On Boolean decision trees with faulty nodes*, Random Structures and Algorithms, 5(3):453-464, 1994.

[16] E. Kushilevitz and Y. Mansour, *Computation in Noisy Radio Networks*, Siam J. on Discrete Math 19(1), 96-108, 2005.

[17] A. V. Kuznetsov, *Information storage in memory assembled from unreliable components*, Problems of Information Transmission, 9, 254-264, 1973.

[18] V. I. Levenšteĭn, *Upper bounds for codes with a fixed weight of vectors* (in Russian), Problemy Peredači Informacii, 7, 3-12, 1971.

[19] J. von Neumann, *Probabilistic logics and the synthesis of reliable organisms for unreliable components*, **Automata Studies**, C. E. Shannon and J. McCarthy Eds., Princeton University Press, 329-378, 1956.

[20] N. Pippenger, *On Networks of Noisy Gates*, Proceedings 26th IEEE Symp. Foundations of Computer Science (FOCS), 30-38, 1985.

[21] R. Reischuk and B. Schmeltz, *Reliable Computation with Noisy Circuits and Decision Trees-A General nlog n Lower Bound*, Proceedings 32th IEEE Symp. Foundations of Computer Science (FOCS), 602-611, 1991.

[22] R. Rajagopalan and L. J. Schulman, *A coding theorem for distributed computation*, 26th Annual ACM Symp. Theory of Computing (STOC), 790-799, 1994.

[23] L. J. Schulman, *Coding for interactive communication*, IEEE Transactions on Information Theory 42(6), 1745-1756, 1996.

[24] D. A. Spielman, *Highly fault tolerant parallel computation*, Proceedings 37th IEEE Symp. Foundations of Computer Science (FOCS), 154-163, 1996.

[25] M. J. Taylor, *Reliable information storage in memories designed from unreliable components*, Bell System Technical Journal, 47, 2299-2337, 1968.

[26] J. H. van Lint, **Introduction to coding theory**, Vol. 86, graduate texts in mathematics, Springer-Verlag, Berlin, 3rd edition, 1999.

[27] I. Wegener, **The complexity of Boolean functions**, John Wiley and Sons, 2001.

[28] A. C. Yao, *On the complexity of communication under noise*, Invited talk the the 5th ISTCS Conf. 1997.