Complexity of Boolean Functions on PRAMs – Lower Bound Techniques*

Mirosław Kutyłowski†

Abstract

Determining time necessary for computing important functions on parallel machines is one of the most important problems in complexity theory for parallel algorithms. Recently, a substantial progress has been made in this area. In this survey paper, we discuss the results that have been obtained for three types of parallel random access machines (PRAMs): CREW, ROBUST and EREW.

1 Introduction

Parallel random access machine (PRAM) is a most abstract model of parallel computers, where interprocessor communication is realized using a shared memory. Each processor of a PRAM can access any cell of a shared memory in one computation step. This is certainly an unrealistic assumption, but it makes the analysis of the parallel algorithms much easier, and we can can concentrate ourselves on inherent complexity of a given problem. This is a reason, why most parallel algorithms have been described in terms of PRAMs.

Each PRAM consists of a collection of processors and common memory cells. Each computation consists of several computation steps. At each step, the processors perform in parallel the following: a processor can read from at most one memory cell; then it does some internal computing (which is deterministic unless otherwise specified); and finally, it has the possibility to write into a chosen memory cell. The first question that comes to mind is how the memory access conflicts are resolved. Many solutions are possible and they lead to a number of different types of PRAMs. If any number of processors can read from a given memory cell simultaneously, then we call it Concurrent-Read PRAM. The PRAMs that forbid concurrent reads are called Exclusive-Read PRAMs. Similarly, Concurrent-Write PRAM is a PRAM that allows many processors to write into a given memory cell during one computation step. Exclusive-Write PRAMs forbid concurrent write into the same cell during one step. Usually, it requires a very careful design of the algorithm to insure that the concurrent reads or writes do not occur. If during computation of an Exclusive-Read (Exclusive-Write) PRAM a read (write) conflict occurs then the computation is interrupted in an emergency state. There are three

*Supported by DFG Grant ME 872/1–4

†Fachbereich Mathematik-Informatik, Universität-GH Paderborn, Postfach 1621, D-W-4790 Paderborn, Germany. Email: mirekk@uni-paderborn.de

main types of PRAMs considered: Exclusive-Read Exclusive-Write (EREW) PRAMs, Concurrent-Read Exclusive-Write (CREW) PRAMs, and Concurrent-Read Concurrent-Write (CRCW) PRAMs. In a case of a CRCW PRAM, one must define value that is written if more than one processor attempts to write into the same cell at the same time. Therefore, many different types of CRCW PRAMs have been defined (COMMON, TOLERANT, COLLISION, ARBITRARY, PRIORITY, ...).

If a PRAM M computes a function f of n arguments, then the arguments are stored initially in the first n memory cells (one argument per cell). The result is given at the last moment of the computation in some specially chosen memory cell.

We investigate time necessary for computing functions on PRAMs. For the sake of simplicity we consider Boolean functions. Each Boolean function f is a function over a finite domain $\{0,1\}^n$ for some $n \in I\!N$, $f:\{0,1\}^n \to \{0,1\}$. We say that time complexity of a function f within a family of PRAMs $\mathcal{A}$ is at most t if there is a PRAM $R \in \mathcal{A}$ that computes f and R performs at most t steps for every input string. Since we consider each finite function separately, the time bounds that we consider are called nonuniform.

The overall strategy of determining lower bounds for Boolean functions on PRAMs is the following: First a complexity measure m for Boolean functions is defined. We require that $m(f)$ should be relatively easy to compute for each Boolean function f. Then the time complexity of f in a given class of PRAMs is expressed in terms of $m(f)$. Of course, such a connection between a complexity measure and time complexities on PRAMs is sometimes difficult to find. Nevertheless, most results that are known are based on this idea.

2 Concurrent-Read Exclusive-Write PRAMs

The first impression is that functions like $\mathrm{OR}_n(x_1,\ldots,x_n) = x_1 \vee \ldots \vee x_n$ cannot be computed in less than $\log n$[†] steps. However, it is not true as shown by Cook, Dwork and Reischuk [4]:

Algorithm 2.1 *There is an n-processor CREW PRAM that computes OR_n in approximately $0.72 \log n$ steps.*

The basic observation is that it is possible to compute OR_2 by a single write operation. Suppose that processor P knows a value $x \in \{0,1\}$ and a cell C stores a value $y \in \{0,1\}$. Then P writes into C if and only if $x = 1$, and the value written is 1. It is easy to see that after writing C always contains $x \vee y$. The algorithm of Cook, Dwork and Reischuk runs so that the processors compute the logical OR for larger and larger groups of the input arguments. Coalescing such groups is done through reading (a processor knowing the OR of one group reads a cell that stores the value of the OR of an other group, and computes the OR of the union of both groups) and through writing (a processor knowing that the value of the OR for a group is equal to 1 writes 1 into a cell storing the value of the OR of another group, and thereby computes the OR for the union of both groups even if it does not write). Applying this mechanism, it is possible to organize the algorithm so that, after step t, each processor knows the value of the OR for a group of F_{2t} input arguments,

[†]throughout the paper, log stands for log to the base 2

and each cell stores the value of the OR of a group of F_{2t+1} arguments, where F_j denotes a Fibonacci number: $F_0 = 0$, $F_1 = 1$ and $F_{j+2} = F_{j+1} + F_j$, for $j \geq 0$. Therefore the above algorithm requires $\phi(n) := \min\{t : F_{2t+1} \geq n\}$ steps to compute OR_n.

2.1 Lower bounds for Boolean functions

2.1.1 Critical complexity lower bounds

The first complexity measure relating Boolean functions with their time complexities on CREW PRAMs was *critical complexity* used by Cook, Dwork and Reischuk [4]. It was later generalized by Nisan [19] to *block-critical complexity*. If f is a function of n arguments then the block-critical complexity of f, denoted $bc(f)$, is the maximum of the numbers

$$\max\{l \mid \exists S_1, \ldots, S_l \subseteq \{1, \ldots, n\} \text{ disjoint s. t. } f(\vec{a}^{S_j}) \neq f(\vec{a}), \text{ for } 1 \leq j \leq l\}$$

taken over all inputs $\vec{a}$, where $\vec{a}^S$ is obtained from $\vec{a}$ by flipping all bits in positions $i \in S$. If each S_i is to contain exactly one element then we get a definition of $c(f)$, *critical complexity* of f. Cook, Dwork and Reischuk [4] prove the following lower bound (the formulation in terms of $bc(f)$ is due to Nisan, the constants presented are due to Parberry and Yan [20]):

Theorem 2.1 *Let f be a Boolean function. Every CREW PRAM computing f makes at least $0.5 \log(bc(f))$ steps.*

Interestingly, there is a tight upper bound shown by Nisan [19]:

Theorem 2.2 *Every Boolean function f can be computed by a CREW PRAM running in time $\approx 2.88 \log(bc(f))$.*

Now we sketch the most important ideas of the proof of Theorem 2.1. We consider an input $\vec{a_0}$ and a set $I \subseteq \{1, \ldots n\}$ such that for every $i \in I$, $f(\vec{a_0}) \neq f(\vec{a_0}(i))$ where $\vec{a_0}(i)$ is the string obtained from $\vec{a_0}$ by flipping the ith bit (so we work with $c(f)$ instead of $bc(f)$). Now let us consider a machine computing f. We say that index i *influences* a processor P at step t on input $\vec{a}$ if the state of P at step t on input $\vec{a}$ differs from the state of P at step t on input $\vec{a}(i)$. Similarly we define indexes influencing the cells. Let $K(P, t, \vec{a})$ and $L(M, t, \vec{a})$ be the set of indexes influencing processor P and cell M respectively, at step t on input $\vec{a}$. The proof estimates cardinalities of the sets $K(P, t, \vec{a})$ and $L(M, t, \vec{a})$ as a function of t. Since the output cell C after the last step T on input $\vec{a_0}$ is influenced by at least $|I|$ indexes, we get immediately a bound on T. Let K_t be the maximal cardinality of the set $K(P, t, \vec{a})$ over all inputs $\vec{a}$ and processors P. Similarly we define L_t. Cook, Dwork and Reischuk show by induction on t that the sizes of K_t and L_t grow at most exponentially with t. It is quite easy to see that $K_{t+1} \leq K_t + L_t$. The difficult part is to express L_{t+1} in terms of K_{t+1} and L_t. The problem is to estimate the number of indexes influencing a cell M if *no processor writes* into M at step $t + 1$ on input $\vec{a}$. The indexes influencing M are these indexes that influence M at step t on $\vec{a}$ (at most L_t indexes), and these indexes u that make a processor write into M at step $t + 1$ on $\vec{a}(u)$. Let $U = \{u_1, \ldots, u_r\}$ be the set of the later indexes. For $u \in U$, by P_{z_u} we denote the processor that writes into M at step $t + 1$ on $\vec{a}(u)$. Let e be the number of pairs (i, j) such that $i, j \in U$ and $P_{z_i} \neq P_{z_j}$.

First we prove that $e \leq r \cdot K_{t+1}$. Consider $i, j \in U$ such that $P_{z_i} \neq P_{z_j}$. If $j \notin K(P_{z_i}, t+1, \vec{a}(i))$ then P_{z_i} writes into M at step $t+1$ on $\vec{a}(i)(j)$. Similarly, if $i \notin K(P_{z_j}, t+1, \vec{a}(j))$ then P_{z_j} writes into M at step $t+1$ on $\vec{a}(j)(i) = \vec{a}(i)(j)$. It follows that either $j \in K(P_{z_i}, t+1, \vec{a}(i))$ or $i \in K(P_{z_j}, t+1, \vec{a}(j))$ since no write conflict may occur. The number of pairs (i, j) such that $i \in K(P_{z_j}, t+1, \vec{a}(j))$ is at most $r \cdot K_{t+1}$, hence $e \leq r \cdot K_{t+1}$.

Now we show that $e \geq r(r - K_{t+1})/2$. Clearly for each $j \in U$, index j influences P_{z_j} on input $\vec{a}$. We take an $i \in U$. If $P_{z_j} = P_{z_i}$, then j influences P_{z_i}. There are at most K_{t+1} indexes j influencing P_{z_i}. Hence there are at least $r - K_{t+1}$ indexes j such that $P_{z_i} \neq P_{z_j}$. Summing up over all i, and dividing by two, since each pair has been considered twice, gives $e \geq r(r - K_{t+1})/2$. By the inequalities obtained, we get $r \cdot K_{t+1} \geq r(r - K_{t+1})/2$, that is, $3K_{t+1} \geq r$. Hence $L_{t+1} \leq L_t + 3K_{t+1}$.

2.1.2 Degree complexity lower bounds

The second important complexity measure, after critical complexity, used for lower bound results is the notion of degree. It provides easier and in most cases more precise lower bounds. The results presented in this section originate from the papers [5, 6] by Dietzfelbinger, Kutyłowski and Reischuk and [17] by Kutyłowski.

Each Boolean function of n arguments can be represented by a polynomial with n variables over an arbitrary field $I\!F$. Indeed, one can represent a Boolean function by a Boolean formula with the operators $\wedge$ and $\neg$. Then we replace each expression $L \wedge M$ by $L \cdot M$, and $\neg L$ by $(1 - L)$. Obviously, the polynomial resulting gives value 1 (respectively 0) for an input $\vec{x} \in \{0, 1\}^n$ if and only if the Boolean function has value 1 (respectively 0) on $\vec{x}$. One can easily prove that for a given Boolean function there is exactly one such polynomial, provided that it contains no terms of the form x^j for $j > 1$. By the degree of a polynomial p we mean the size of the biggest monomial occurring in p. For instance, degree of the polynomial $x_1 - x_1x_2x_7 + 2x_3x_2$ is 3, because of the monomial $x_1x_2x_7$ consisting of 3 variables. Throughout this section, we shall consider only polynomials over $\mathbb{Q}$ (polynomials over finite fields are investigated in section 5). For a Boolean function f, let $\deg(f)$ denote the degree of the polynomial over $\mathbb{Q}$ representing f.

Theorem 2.3 *If a Boolean function f is computed by a CREW PRAM in T steps, then $T \geq \phi(\deg(f)) \approx 0.72 \log(\deg(f))$.*

We sketch the proof of this theorem. Each state q of a processor (a cell) P after step t can be characterized by a Boolean function $f_{P,q,t}$ such that for every input $\vec{x}$, $f_{P,q,t}(\vec{x}) = 1$ if and only if processor (cell) P is in state q after step t in the computation on input $\vec{x}$. The crucial observation is that one can estimate the maximal degree of the functions $f_{P,q,t}$ in terms of t. Each processor has a fixed initial state, so $f_{P,q,0}$ is a constant function 0 or 1, for each q and P. These functions have degree 0. If C_i is one of the input cells, then two initial contents of C_i are possible. If x_i is the variable denoting the ith input bit, then these states of C_i are represented by the polynomials x_i and $1 - x_i$ of degree 1. Other contents are not possible and therefore are represented by the constant polynomial 0. We may assume that the processors do not forget information. In this case, each processor state is determined one to one by a sequence of the symbols already read. It follows that if a processor P in state q at step t reads from a cell C that contains a symbol s, then the state

of P changes to the state that corresponds to the polynomial $f_{P,q,t-1} \cdot f_{C,s,t-1}$. The degree of this polynomial is at most the sum of the degrees of $f_{P,q,t-1}$ and $f_{C,s,t-1}$. Now we inspect what happens in the case of a write operation. The important and usually difficult case is when no processor writes at step t into a cell C containing a symbol s. This happens if and only if $f_{C,s,t-1} = 1$ and $f_{P_i,q_i,t} = 0$, for each P_i, q_i such that processor P_i writes in state q_i into C at step t. No concurrent write can ever occur, hence the polynomial $f_{P_1,q_1,t} + f_{P_2,q_2,t} + \ldots + f_{P_l,q_l,t}$ correctly represents the situation "somebody writes into C at step t". So the polynomial $f_{C,s,t}$ is equal to $f_{C,s,t-1} \cdot (1 - (f_{P_1,q_1,t} + f_{P_2,q_2,t} + \ldots + f_{P_l,q_l,t}))$. The degree of $f_{C,s,t}$ is therefore at most the degree of $f_{C,s,t-1}$ plus the maximum of the degrees of $f_{P_1,q_1,t}, f_{P_2,q_2,t}, \ldots, f_{P_l,q_l,t}$.

Summarizing our observations, if p_t denotes the maximal degree of the polynomials $f_{P,q,t}$ over all processors P and states q, and c_t denotes the maximal degree of the polynomials $f_{C,s,t}$ over all cells C and symbols s, then the following holds:

$p_0 = 0, \quad c_0 = 1$;

$p_{i+1} \leq p_i + c_i, \quad c_{i+1} \leq p_{i+1} + c_i \qquad$ for $i \geq 0$.

It follows that $p_t \leq F_{2t}$ and $c_t \leq F_{2t+1}$. If a CREW PRAM computes a function f in T steps, then after step T the output cell C stores the value of f. Hence $f(\vec{x}) = 1$ if and only if $f_{C,1,T}(\vec{x}) = 1$, that is, $f = f_{C,1,T}$. So $\deg(f) \leq F_{2T+1}$, and $T \geq \phi(\deg(f))$. This completes the proof of Theorem 2.3.

In many cases, determining the degree of a Boolean function is not difficult. We can make the things even easier as shown by the following theorems:

Theorem 2.4 *If f is a Boolean function of n arguments and $|f^{-1}(1)| = 2^i \cdot u$ for u odd, then each CREW PRAM computing f makes at least $\phi(n-i) \approx 0.72 \log(n-i)$ steps.*

Let $\mathrm{PARITY}_n(x_1, \ldots, x_n) = \sum x_i \pmod 2$.

Theorem 2.5 *If f is a Boolean function of n arguments and*

$$\left|\left\{ \vec{a} \mid f(\vec{a}) = 1 \wedge \mathrm{PARITY}_n(\vec{a}) = 0 \right\}\right| \neq \left|\left\{ \vec{a} \mid f(\vec{a}) = 1 \wedge \mathrm{PARITY}_n(\vec{a}) = 1 \right\}\right|,$$

then each CREW PRAM computing f makes at least $\phi(n) \approx 0.72 \log(n)$ steps.

For the first theorem, one can show that if $|f^{-1}(1)| = 2^i \cdot u$ for u odd, then $deg(f) \leq n - i$. The second theorem follows from the fact that the coefficient of the monomial $x_1 x_2 \cdots x_n$ in the polynomial representing f is equal to $\pm \sum_{\vec{a} \in f^{-1}(1)} (-1)^{PARITY_n(\vec{a})}$.

2.1.3 Matching upper bounds

Since most Boolean functions of n arguments have degree n [6], $\phi(n) \approx 0.72 \log n$ is the lower bound for most Boolean functions. On the other hand, each Boolean function of n arguments can be computed in time $\phi(n) + 1$ by a CREW PRAM with $n \cdot 2^n$ processors. This can be done as follows (for a precise description see [17]): for each possible string $s = [s_1, \ldots, s_n]$ of n bits, there is a group of n processors G_s that in one step copies the input string into cells $C_{s,1}, \ldots, C_{s,n}$. Then for each $s_i = 1$, the content of the cell $C_{s,i}$ is changed from 1 to 0 or from 0 to 1. Next, G_s computes the OR_n of the bits stored in $C_{s,1}, \ldots, C_{s,n}$. This is 0 if and only if the input string is equal to s. Hence for an input string s, only the group G_s gets 0 as the result. Then the first processor of G_s writes the value of the function on s into the output cell.

It is interesting to see that for many important functions, the computation time $\phi(n)$ can be almost achieved with only n processors, as shown by Dietzfelbinger, Kutyłowski and Reischuk [7]:

Theorem 2.6 *The following can be performed by n-processor CREW PRAMs in time $\phi(n) + o(\log n)$:*

- *evaluating Boolean formulas of size n and depth $\log n$,*
- *computing symmetric functions of n bits,*
- *computing parallel prefix for a product of n arguments for each associative operation over a k-valued domain for $k = o(\sqrt{\log n})$ (hence for instance, adding two n-bit binary numbers),*
- *sorting strings of n bits.*

There is a rich variety of techniques used by the above algorithms. We sketch only one method that is quite frequently used. Consider for example the function PARITY_n. The CREW PRAM computation that we describe consists of several stages. With the beginning of each stage i we associate values $y_{i,1}, \ldots, y_{i,s(i)}$ such that $\text{PARITY}_{s(i)}(y_{i,1}, \ldots, y_{i,s(i)})$ is equal to the PARITY of the input string. The number $s(i)$ is getting smaller after each stage, until finally $s(i) = 1$ after the last stage. Let $k = k(i) = \max\{j \mid n/s(i) \geq 2^j\}$. Hence each variable $y_{i,j}$ corresponds to at least 2^k processors. The stage i is performed in the following way: Divide the values $y_{i,1}, \ldots, y_{i,s(i)}$ into disjoint groups of k elements, each group corresponding to $k \cdot 2^k$ processors. Then compute the PARITY of each group using the time optimal algorithm described above. The results are taken as the values $y_{i+1,1}, \ldots, y_{i+1,s(i+1)}$ used by the next stage. Hence $s(i+1) = s(i)/k(i)$. Note that $n/s(i+1)$ is bigger than $n/s(i)$ and therefore the groups considered for stage $i+1$ are bigger than these of stage i. Such dynamic incrementing the size of the groups leads to $L = O(\log(n)/\log\log(n))$ stages of the above algorithm. For each stage, few extra steps are necessary, hence together the algorithm runs in time

$$\begin{aligned}\sum_{i=1}^{L}(\phi(k(i)) + 1 + c) &\approx \sum_{i=1}^{L} 0.72\log(k(i)) + O(L) = 0.72\log(\textstyle\prod k(i)) + O(L) \\ &= 0.72\log(n) + O(L) = 0.72\log(n) + O(\log(n)/\log\log(n)).\end{aligned}$$

2.2 CREW PRAMs working on restricted domains

So far we have considered Boolean functions on domains $\{0,1\}^n$, $n \in I\!N$. The situation might be quite different if a domain is only a subset of $\{0,1\}^n$. We discuss it on the example of the *k-compaction problem*. The inputs for k-compaction problem are strings containing at most k ones and otherwise 0's. The output is a set of all positions in the input string where the 1's are stored. Hagerup and Nowak [14] show tight bounds for solving k-compaction problem:

Theorem 2.7 *When no restrictions are placed on the number of processors, k-compaction problem for strings of length n can be solved by a CREW PRAM in time $O(\min\{k, \log n\})$, and at least $\Omega(\min\{k, \log n\})$ steps are necessary.*

The upper bound is straightforward. The proof of the lower bound uses similar technique as introduced in section 2.1.1. On the other hand, a direct application of the methods based on degree complexity yields only a lower bound $\Omega(\log k)$.

Note that 2-compaction problem for strings of length n can be solved in a constant time by an n^2-processor CREW PRAM, and in $O(\log\log n)$ steps by an n-processor CREW PRAM.

For the first algorithm, for every pair of input bits there is a processor testing if they are different from 0. The single processor that encounters two values different from 0 may write the result into the output cell without causing a write conflict. If no processor writes, then there is at most one 1 in the input string. Then for each $i \leq n$, processor P_i reads the ith bit and if it is not equal to 0, then P_i writes i into the output cell. For the second algorithm, divide the input into $\sqrt{n}$ substrings of length $\sqrt{n}$ and solve 2-compaction problem for each substring using $\sqrt{n}$ processors. There are $\sqrt{n}$ results, with at most 2 of them nonempty. Applying the first algorithm, one can find these nonempty results in a constant number of steps with $(\sqrt{n})^2 = n$ processors.

Fich and Ragde [9] prove that this algorithm is optimal.

Theorem 2.8 *Solving 2-compaction problem for strings of length n requires more than $(\log\log n - \log\log(5p/n))/2$ steps of a CREW PRAM with $p \geq n$ processors.*

The idea of the proof is the following. For each step t, a set V_t of input variables is constructed with $|V_t| \geq n^{2^{2t}}/(5p)^{2^{2t}-1}$. The input strings with all variables outside V_t set to 0 have the property that after step t a state of each processor and memory cell depends on at most one variable of V_t. During the next step, a processor knowing one variable of V_t may read a cell knowing another variable of V_t. After reading, the processor state depends on both variables. A similar effect may occur after writing. To find V_{t+1} we define a graph over V_t with an edge between each x_i and x_j such that a processor knowing x_i reads from (writes into) a cell knowing x_j. V_{t+1} is chosen as an independent subset of V_t of the maximal size. By a theorem of Turan, this set is big enough to get the required approximation of $|V_{t+1}|$. Clearly, if a computation terminates after T steps, then $|V_T| = 1$. Hence, $1 \geq n^{2^{2T}}/(5p)^{2^{2T}-1}$ and Theorem 2.8 follows.

2.3 Open problems for CREW PRAMs

The lower bounds yield by critical complexity and degree are independent from the number of processors used by a PRAM. More precise methods, taking the number of processors into account, are still not known. Few upper bounds for the bounded number of processors have been presented in section 2.1.3.

The following versions of CREW PRAMs may be considered:

Oblivious: for each step t and cell C, either no processor ever writes into C at step t or some fixed processor P writes into C at step t, for every input string.

Always writing: for each step t and cell C, either no processor ever writes into C at step t or for each input string there is a processor that writes into C at step t.

Owner write: each cell has its "owner", the only processor that might write into this cell.

Nonoblivious: the model without any restrictions.

Nisan shows that for CREW PRAMs with an unbounded number of processors and cells, these models are equivalent [19]. More precisely, he proves that for each Boolean function the computation times in these models differ by only a constant factor (sometimes they really differ by a constant factor). It is not known whether it is true for CREW PRAMs with a bounded number of processors. A positive answer might simplify the search for lower bounds for CREW PRAMs with a bounded number of processors.

The next important step to make CREW PRAMs more realistic is to assume that memory cells can store binary words of a fixed size that is called *wordsize*. Restricting simultaneously the wordsize and the number of processors may lead to a dramatical increase of computation time, as shown by Bellantoni [3]. Namely, computing most Boolean functions of n arguments takes at least $\frac{n}{w} - 2\log p - 9$ steps on CRCW PRAMs with p processors and wordsize w. This result follows from a counting argument and says nothing about the functions that are difficult to compute on such PRAMs.

3 Nondeterministic CREW PRAMs

At each step, a processor of a nondeterministic CREW PRAM may have different alternatives how to proceed, but no matter how the individual processors behave, no write conflicts are allowed to occur. A function f is computed by such a machine if for all inputs $\vec{a}$ with $f(\vec{a}) = 1$ there is at least one computation that yields the output 1, and for $\vec{a}$ with $f(\vec{a}) = 0$ there is no such computation. Note that the OR can be computed in a constant time in this model. So, the bounds of section 2.1.1 and 2.1.2 do not apply to nondeterministic CREW PRAMs. Quite easily, one can show the following lemma:

> If a Boolean function f can be computed by a nondeterministic CREW PRAM in time T, then $f = \bigvee_{g\in\Gamma} g$ for a set Γ of Boolean functions where each $g \in \Gamma$ can be computed by a CREW PRAM (deterministic) in time T.

Suppose that f is a Boolean function such that for each function g, $0 \neq g \leq f$, computing g on a CREW PRAM requires at least T steps. Then, by the above lemma, each nondeterministic CREW PRAM computing f runs in time at least T. There are many functions having this property. For example, if $0 \neq g \leq \mathrm{PARITY}_n$, then by Theorem 2.5, computing g takes at least $\phi(n)$ CREW PRAM steps. Similarly, if f is a function of n arguments and $|f^{-1}(1)| \leq 2^k$, then for each g, $0 \neq g \leq f$, we get $|g^{-1}(1)| \leq 2^k$, and by Theorem 2.4, computing g takes at least $\phi(n-k)$ steps of a CREW PRAM. Therefore, we obtain the following theorem due to Dietzfelbinger, Kutyłowski and Reischuk [5, 6]:

Theorem 3.1 *(a) If $0 \neq f \leq \mathrm{PARITY}_n$, then every nondeterministic CREW PRAM computing f makes at least $\phi(n) \approx 0.72 \log n$ steps.*

(b) If $0 \neq f$ is a Boolean function of n arguments and $k = \lfloor \log |f^{-1}(1)| \rfloor$, then every nondeterministic CREW PRAM computing f makes at least $\phi(n-k) \approx 0.72 \log(n-k)$ steps.

4 Probabilistic CREW PRAMs

At each step, a processor of a probabilistic CREW PRAM decides by means of a random experiment what computation path to follow. The probability of a write conflict has to be zero. The machine must yield the correct results with probability greater than $\frac{1}{2}$ (unbounded error case) after a given number of steps. We consider also a bounded-error model, where the correct result must be given with probability at least $\frac{1}{2}+\epsilon$ for a fixed ϵ.

4.1 Unbounded-error probabilistic CREW PRAMs

One can easily define a probabilistic CREW PRAM that computes OR_n in one step with probability greater than $\frac{1}{2}$. Simply, a processor chooses randomly $i \leq n$ and examines the ith input bit. If it is 1, then the answer is 1. If it is 0, then the machine answers 0 with probability $\frac{2n-1}{4(n-1)} > \frac{1}{2}$, and 1 otherwise. It is easy to see that if the input string contains a 1, then the machine answers 1 with probability at least $\frac{1}{n} + \frac{n-1}{n} \cdot \frac{2n-3}{4(n-1)} = \frac{2n+1}{4n} > \frac{1}{2}$.

Surprisingly, Dietzfelbinger, Kutyłowski and Reischuk [5, 6] show that PARITY_n cannot be computed by a probabilistic CREW PRAM with unbounded error faster than by deterministic CREW PRAMs.

Theorem 4.1 *If a probabilistic CREW PRAM computes* PARITY_n *in* T *steps (with unbounded error), then* $T \geq \phi(n) \approx 0.72 \log(n)$.

We briefly sketch the proof. For each probabilistic CREW PRAM there is a probabilistic CREW PRAM that makes only one random guess (from a finite domain) at the beginning of the computation and yields the results with the same probabilities after the same number of steps. Each of these guesses defines a deterministic CREW PRAM, and therefore a Boolean function computed by this machine. Let M be such a probabilistic CREW PRAM computing PARITY_n in T steps, and Γ be the family of functions defined by the random guesses for M. For $g \in \Gamma$, let p_g be the probability that M chooses a machine computing g. Define a real-valued function $h = \sum_{g \in \Gamma} p_g \cdot g$. It can be easily checked that for all $\vec{a} \in \{0,1\}^n$:

$$\mathrm{Prob}(M \text{ outputs } 1 - \mathrm{PARITY}_n(\vec{a}) \text{ on input } \vec{a}) = |h(\vec{a}) - \mathrm{PARITY}_n(\vec{a})|.$$

Hence $h(\vec{a}) < \frac{1}{2}$ for all $\vec{a}$ of even parity and $h(\vec{a}) > \frac{1}{2}$ for all $\vec{a}$ of odd parity.

Since each $g \in \Gamma$ can be computed in T steps, $\deg(g) \leq F_{2T+1}$. Hence $\deg(h) \leq F_{2T+1}$, too. On the other hand, one may show that the coefficient of the monomial $x_1 x_2 \cdots x_n$ in the polynomial defining h is equal to:

$$\pm \sum_{\vec{a} \in \{0,1\}^n} h(\vec{a}) \cdot (-1)^{\mathrm{PARITY}_n(\vec{a})} = \pm \left(\sum_{\vec{a} \in \mathrm{PARITY}_n^{-1}(0)} h(\vec{a}) - \sum_{\vec{a} \in \mathrm{PARITY}_n^{-1}(1)} h(\vec{a}) \right).$$

The first sum is greater than $\frac{1}{2} \cdot 2^{n-1}$, the second sum is smaller than $\frac{1}{2} \cdot 2^{n-1}$. Hence their difference cannot be equal to 0. So $\deg(h) = n$ and therefore $T = \phi(n)$.

4.2 Bounded-error probabilistic CREW PRAMs

Using a result of Szegedy [21], Dietzfelbinger, Kutyłowski and Reischuk [5, 6] show that in the case of the bounded-error model, probabilistic CREW PRAMs are not significantly faster than deterministic CREW PRAMs (this result follows also from the paper of Nisan [19], however this yields a constant factor bigger than 8).

Theorem 4.2 *The probabilistic CREW PRAM time complexity of a Boolean function in the bounded-error model differs from the deterministic CREW PRAM time complexity at most by a factor of 8 and an additive term depending on the error ϵ.*

The key lemma leading to this result is the following:

> *If a Boolean function f is computed by a probabilistic CREW PRAM in T steps with bounded error ϵ, then $T \geq \phi(\sqrt{\epsilon \cdot bc(f)})$.*

5 ROBUST PRAMs

So far, we have not considered PRAMs that allow concurrent writes (so called CRCW PRAMs). It is easy to see that concurrent write capability adds much to computational power of a PRAM. For instance, OR_n can be computed in one step by a CRCW PRAM: First, the input bits are read in parallel by n processors. Then the processors that have read 1 write 1 into the output cell. This is a correct algorithm for COMMON CRCW PRAMs where in a case of a write conflict all processors must attempt to write the same symbol and the symbol actually written coincides with the symbol that the processors attempt to write. The speed-up against CREW PRAMs is dramatic: from $\Theta(\log n)$ to 1 step. However, we are in a typical annoying situation. In order to construct a correct algorithm, we have had to determine precisely how the machine behaves in a case of a write conflict. There are many reasonable ways of resolving write conflicts leading to many different types of CRCW PRAMs. Hagerup and Radzik [15] proposed the most general model of CRCW PRAMs, which they call a ROBUST PRAM. They assume that in a case of a write conflict the symbol actually written is chosen nondeterministically from the set of **all** symbols that are used by the machine. However, an algorithm must compute the correct answer no matter what values appear when write conflicts occur. By this assumption, such an algorithm is working correctly on any CRCW PRAM, in other words, it is *robust* against machine type. This is a nice property, but, as we shall see in the next subsection, the price that we have to pay is unacceptable. Namely, computation times on ROBUST PRAMs are almost the same as in the case of CREW PRAMs. This means that ROBUST PRAMs are significantly slower than other CRCW PRAMs.

5.1 Degree complexity for computations on ROBUST PRAMs

In this subsection we sketch the results obtained by Fich, Impagliazzo, Kapron and King [8] and Kutyłowski [18]. In order to show a lower time bound T for computing a function f on ROBUST PRAMs it suffices to prove that some CRCW PRAMs with specially chosen

way of resolving write conflicts require at least T steps for computing f. This will be our overall strategy.

We start with machines of wordsize 1, where each memory cell may store only one bit. Consider the following way of resolving write conflicts. If a cell C contains $h \in \{0,1\}$ and during a given step n_0 processors attempt to write 0 into C and n_1 processors attempt to write 1 into C, then the value written into C is

$$h + (0-h) \cdot n_0 + (1-h) \cdot n_1 \pmod 2.$$

It is easy to see that this definition is consistent with the case when at most one processor writes. The states of the processors and memory cells of the resulting machine can be described by Boolean functions, as in the case of the CREW PRAM. Each Boolean function can be represented by a polynomial over the field $\mathbb{F}_2 = \{0,1\}$. This representation is well suited for these machines because of the way of concurrent writing (summation modulo 2 corresponds to addition in $\mathbb{F}_2$).

We have to estimate complexity of polynomials over $\mathbb{F}_2$ representing states of processors and memory cells after a given number of steps. As complexity measure we take degree, denoted now by $\deg_2$ to indicate that we consider polynomials over $\mathbb{F}_2$. We check that degrees of the polynomials representing cell states approximately double during one step. The important case is again the case of writing. The polynomial describing the state of cell C in which it contains 1 after step t is

$$H(\vec{x}) = h(\vec{x}) + (0-h(\vec{x})) \cdot \Big(f_0(\vec{x}) + \ldots + f_l(\vec{x})\Big) + (1-h(\vec{x})) \cdot \Big(g_0(\vec{x}) + \ldots + g_m(\vec{x})\Big) \quad \text{(over } \mathbb{F}_2\text{)},$$

where polynomial h represents the state in which C contains 1 after step $t-1$, $f_0, \ldots, f_l$ and $g_0, \ldots, g_m$ represent the states of processors that cause writing 0, respectively 1, into C at step t. Then $\deg_2(H)$ is bounded by $\deg_2(h) + \max\{\deg_2(w) \mid w \in \{f_0, \ldots, f_l\} \cup \{g_0, \ldots, g_m\}\}$. The polynomial representing the state of C after step t in which C contains 0 is $1 - H(\vec{x})$, that is, of the same degree as $H(\vec{x})$. More elaborate counting shows that after step t degrees of the polynomials representing states of memory cells do not exceed F_{2t+1}.

If such a machine computing function f halts after step T, then the output cell contains 1 if and only if f has value 1 on the given input. Therefore $\deg(f) \leq F_{2T+1}$. Let $\mathrm{ROBUST}_w(f)$ denote the number of steps necessary to compute a Boolean function f on ROBUST PRAMs of wordsize w. Thus we have obtained the following theorem:

Theorem 5.1 *For every Boolean function f,*

$$\mathrm{ROBUST}_1(f) \geq \phi(\deg_2(f)) \approx 0.72 \log(\deg_2(f)).$$

In particular, $\mathrm{ROBUST}_1(\mathrm{OR}_n) \geq \phi(n)$.

It is astonishing that, by the above result, computing OR_n requires exactly the same number of steps of a ROBUST PRAM of wordsize 1 as of a CREW PRAM.

In Theorem 5.1, one can replace $\mathbb{F}_2$ by any finite field $\mathbb{F}_p$. Then the time bound is $c \cdot \log(\deg_p(f))$ where $c \approx \frac{1}{\log(2p-1)}$. Thereby, we get lower bounds $\Omega(\log(n))$ for such functions as PARITY_n (it is easy to see that $\deg_2(\mathrm{PARITY}_n) = 1$ but $\deg_3(\mathrm{PARITY}_n) = n$).

The lower bound of Theorem 5.1 holds for an arbitrary number of processors but only for the machines of wordsize 1. However, since one can simulate one step of a ROBUST PRAM of wordsize w by $\log(w)$ steps of a ROBUST PRAM of wordsize 1, we get the following lower bound:

Theorem 5.2 *For every Boolean function f and a prime number q,*

$$\mathrm{ROBUST}_w(f) \geq \Omega\left(\frac{\log(\deg_q(f))}{\log w}\right).$$

We describe the idea of the simulation mentioned above. Each memory cell C of wordsize w is coded by a group G_C of w cells of wordsize 1 so that each bit of a word stored by C is placed in a separate cell of G_C. We describe how to read these w cells in $\log(w)$ steps without using memory cells of wordsize greater than 1 (this is the key point of the simulation). It requires a preprocessing for which we use 2^w groups of w processors each. Each group P_s, named by a binary string s of length w, checks whether the cells of G_C store string s. It can be done in $\log(w)$ steps, for instance by the algorithm of Cook, Dwork and Reischuk [4]. Exactly one group P_s gets a positive answer. Then P_s encodes the content of C in a different way. 2^w cells containing initially 0's are used. The first processor of P_s writes 1 into one of these cells, namely the cell with the index s. Then we are ready for simulation of reading from C. For each processor R of the original machine we have a large set of processors that simulate R. If R reads from C, then these processors read in parallel all 2^w cells that code the content of C. The processors that read the cell storing 1 get full information about the content of C and can further perform the simulation. The other processors fail and are not used anymore.

Theorem 5.2 provides good lower bounds for reasonable wordsizes such as $\log(n)$. There is a lower bound that holds for every wordsize provided that the number of processors is "realistic":

Theorem 5.3 *Let f be a Boolean function. If an n-processor ROBUST PRAM (of an arbitrary wordsize) computes f in T steps, and $\log(\deg_2(f)) \geq (\log\log n)^2$, then*

$$T = \Omega(\sqrt{\log(\deg_2(f))}\,).$$

The proof is based on the observation that during t steps an n-processor ROBUST PRAM really uses only about n^{2^t} symbols in the common memory. It corresponds to wordsize $\log(n) \cdot 2^t$ and can be simulated with a delay $\log\log(n) + t$ by a ROBUST PRAM of wordsize 1. Hence, if f is computed in T steps by an n-processor ROBUST PRAM, then there is a ROBUST PRAM of wordsize 1 that computes f in $T \cdot (\log\log(n) + T)$ steps. So

$$T \cdot (\log\log(n) + T) \geq \Omega(\log(\deg_2(f)))$$

and Theorem 5.3 follows.

We say that a CRCW PRAM is deterministic, if the result of a concurrent-write operation is uniquely determined by the state of the machine immediately before writing. All lower bounds obtained so far for ROBUST PRAMs were in fact lower bounds for some special deterministic CRCW PRAMs. The lower bounds of Theorems 5.2 and 5.3 do not cover one case: when a ROBUST PRAM uses a lot of processors and the memory cells

have a big wordsize. This has a deep reason. Fich, Impagliazzo, Kapron and King [8] show that each deterministic CRCW PRAM with 2^n processors and one memory cell of wordsize n can compute OR_n in two steps. We sketch their construction. They consider the following situation: each processor in a given set W attempts to write its number into a fixed cell. The outcome of this operation, denoted by $v(W)$, depends on the set W. By a set theoretical argument, they show that the set of 2^n processors can be partitioned into sets A, N, D with $|D| = n$ such that the following holds:

$$\exists_{S \subseteq D} \forall_{S' \subseteq D} \ (S' \neq S \Rightarrow v(A \cup S) \neq v(A \cup S')) \,.$$

Using this property, OR_n can be computed as follows: Let each input bit be read by one processor in D. Then during the write phase, processors attempt to write their numbers into a fixed cell: processors in A always attempt to write, processors in N never attempt to write, a processor in S attempts to write if it reads 0 from the input, a processor in $D \setminus S$ attempts to write if it reads 1 from the input. It is easy to see that $\mathrm{OR}_n(\vec{x}) = 0$ if and only if the result of writing is $v(A \cup S)$.

5.2 ROBUST PRAMs working on restricted domains

In this section, we show that ROBUST PRAMs may be useful for functions defined over restricted domains. As in section 2.2, we consider the compaction problem. For small k, the k-compaction problem can be solved in $O(\log k)$ time by a ROBUST PRAM with n processors. This is much faster than is possible for CREW PRAMs.

We start with a simple algorithm for the 2-compaction problem. First, we arrange n input cells in a square of size $\sqrt{n} \times \sqrt{n}$. Additionally, we use memory cells $Z_1, \ldots, Z_{\sqrt{n}}$ and $S_1, \ldots, S_{\sqrt{n}}$. Each input cell is read by a processor, and if the cell stores 1, then its name is written into 2 cells. Namely, the processor that reads the input cell lying in the row i and the column j writes (i, j) into Z_i and S_j. Then two 2-compaction problems are considered: one for $Z_1, \ldots, Z_{\sqrt{n}}$ and one for $S_1, \ldots, S_{\sqrt{n}}$, in order to find places where these cells contain symbols different from 0. It can be done in 3 steps by a CREW PRAM with $n = (\sqrt{n})^2$ processors. Note that if the input cells containing 1's do not lie in the same row of the input tableau, then there is no write conflict while writing into $Z_1, \ldots, Z_{\sqrt{n}}$, and $Z_1, \ldots, Z_{\sqrt{n}}$ contain the correct addresses of the input cells that store 1's. Similarly, if these input cells do not lie in the same column, then $S_1, \ldots, S_{\sqrt{n}}$ contain the correct addresses of the input cells that store 1's. It cannot happen that two different cells lie in the same row **and** in the same column, hence at least one of the solutions given by $S_1, \ldots, S_{\sqrt{n}}$ and $Z_1, \ldots, Z_{\sqrt{n}}$ is correct. It is easy to find which one.

The second algorithm, due to Hagerup [12], solves the k-compaction problem for an arbitrary k:

Algorithm 5.1 *Let $k \in \mathbb{N}$. The k-compaction problem for inputs of length n can be solved in $O(\log k)$ time by a ROBUST PRAM with $\sum_{i=1}^{k} \binom{n}{k}$ processors.*

The algorithm uses auxiliary memory cells $A(1), A(2), \ldots, A(k)$ initially containing 0's. For each subset S of the input cells of cardinality at most k, there is a corresponding set of $|S|$ processors. In $O(\log(|S|)) = O(\log k)$ steps, these processors check whether each cell of S contains 1. If it is found true, then the first processor corresponding to S writes

the tuple of $|S|$ addresses of the cells of S into cell $A(|S|)$. If there are exactly l input cells that contain 1's, then after the write step: the cells $A(l+1), \ldots, A(k)$ still contain 0's; the cell $A(l)$ contains the tuple of all l addresses of the input cells that store 1's; the cells $A(1), \ldots, A(l-1)$ have unpredictable contents caused by write conflicts. In $O(\log k)$ steps, the last cell $A(i)$ with a content different from 0 is determined. Then the content of $A(i)$ is copied into the output cell.

Now, we sketch the idea of a ROBUST PRAM algorithm for the k-compaction problem due to Kowaluk and Loryś [16]:

Algorithm 5.2 *Let $k \leq O(\sqrt{\log n})$. The k-compaction problem for inputs of length n can be solved by an n-processor ROBUST PRAM running in $O(\log k)$ time.*

(The authors claim that the algorithm still works for larger k's.) We describe only the most important features of the algorithm. The general idea is to write the addresses of the input cells that store 1's into a relatively small tableau of cells. It should be written so that each address of an input cell storing 1 occurs at least once in the tableau (which is difficult) and the tableau contains no garbage (which is easy). Then the addresses stored in the tableau are collected. This is possible, since the size of the tableau is small relative to the size of the input string n, and we can use n processors.

Let $l = 2k \cdot (k-1)$. By a classical result from number theory, one can show that in the interval $[n^{\frac{1}{2k}}, 2 \cdot n^{\frac{1}{2k}}]$ there are at least l prime numbers. We fix l different prime numbers from this interval, and call them $p_1, \ldots, p_l$. The following key property holds:

If $a_1, ..., a_k \in I\!N \cap [1, n]$ are all different, then

$$\forall_{1 \leq j \leq k} \, \exists_{1 \leq i \leq l} \, \forall_{r \neq j} \quad a_j \not\equiv a_r \bmod p_i.$$

The proof is quite easy: For a moment we fix j, $j \leq k$. Let $Z_{jr} = \{p_i \mid i \leq l \text{ and } a_j \equiv a_r \bmod p_i\}$. Clearly, $|Z_{jr}| < 2k$, since $|a_j - a_r| < n$. Hence $|\bigcup_{r \neq j} Z_{jr}| < 2k \cdot (k-1)$, for each j. So, there is $p_i \notin \bigcup_{r \neq j} Z_{jr}$. Then $a_j \not\equiv a_r \bmod p_i$, for each $r \neq j$.

The algorithm uses a tableau with l rows, row i consisting of p_i memory cells $S_i(0), \ldots, S_i(p_i - 1)$. During the first phase of the algorithm, for each input cell j that stores 1, and each $i \leq l$, a processor attempts to write j into cell $S_i(j \bmod p_i)$. (Performing this in $\log(k)$ steps with n processors is not straightforward, but we do not specify more details.) Write conflicts may occur at many places, producing unpredictable results, but by the property shown above, for each cell j storing 1, there is i such that the processor writing j into the cell $S_i(j \bmod p_i)$ is the only processor accessing this cell.

After writing, some cells might be affected by write conflicts. To eliminate any garbage, for each cell C of the tableau storing an $i \neq 0$, it is checked if the ith input cell stores 1. If not, then 0 is written into C.

Now the addresses stored in the tableau are collected. Note that each row contains at most k addresses. The addresses stored in each row are found by Algorithm 5.1. The number of the processors used is at most

$$l \cdot \sum_{i=1}^{k} \binom{2 \cdot n^{\frac{1}{2k}}}{i} \leq l \cdot (2 \cdot n^{\frac{1}{2k}})^k \leq n.$$

Concatenating the sets of addresses from l rows can be done in a straightforward way in $O(\log l) = O(\log k)$ steps.

5.3 Probabilistic ROBUST PRAMs

By the results of section 5.1, ROBUST PRAMs have almost the same computational power as CREW PRAMs. This is no longer true for probabilistic ROBUST PRAMs, as shown by Hagerup and Radzik [15]. They prove that even such powerful CRCW PRAMs like PRIORITY PRAMs can be efficiently simulated by probabilistic ROBUST PRAMs:

Theorem 5.4 *Let $\epsilon, \beta \in \mathbb{R}$ be fixed. One step of an n-processor PRIORITY PRAM can be simulated with probability at least $1 - n^{-(\log n)^\beta}$ by $\log\log(n)$ steps of a probabilistic ROBUST PRAM that uses at most $n(\log n)^{1+\epsilon}$ processors .*

We sketch the technique leading to a similar simulation for ARBITRARY PRAMs. An ARBITRARY PRAM, in a case of a write conflict, allows one processor (which can be arbitrary) to write what it wants. To perform a simulation of an ARBITRARY by a probabilistic ROBUST PRAM, for each memory cell C where a write conflict occurs, we must randomly choose a single processor that attempts to write into C. Moreover, the probability of the success must be high. For a cell C, we consider an array U_C of $\log(n)$ rows, each row consisting of $e \cdot \log(n)$ cells (e is a constant required for technical reasons). If a processor P wants to write into C, then, during the simulation, P chooses i, for $i = 1, \ldots, \log(n)$, being chosen with probability 2^{-i} (with probability $2^{-\log(n)}$, the processor chooses no row and remains inactive). Then P chooses a number j from the uniform distribution over $\{1, \ldots, \log(n)\}$. Finally, P attempts to write its name into the jth cell in row i of U_C. We say that P is successful if, after writing, the name stored in this cell is the name of P. Because of a possible write conflict, it may happen that this cell stores something that is not the name of a processor that has attempted to write its name there. However, if exactly one processors writes into a cell, then this processor is successful. (The opposite may not be true.)

One can prove that if m processors of the original ARBITRARY PRAM attempt to write into cell C and

$$c \cdot 2^{i-1} \cdot \log(n) \leq m \leq c \cdot 2^i \cdot \log(n)$$

($c \in \mathbb{N}$), then the number of the simulating processors that write into row i of U_C is $d \cdot \Theta(\log(n))$ with probability $1 - n^{d \cdot \Omega(1)}$. Moreover, at least half of these processors will be successful with probability $1 - n^{-d \cdot \Omega(1)}$, if the constants are appropriately chosen. It is easy to mark the names of successful processors written into U_C. Then, for each successful processor, it is checked if its name is stored in the first marked cell of U_C. Since U_C contains $e \cdot (\log(n))^2$ cells, this can be done by a CREW PRAM in $\Omega(\log\log(n))$ time.

6 Exclusive-Read Exclusive-Write PRAMs

In this section, we consider the EREW PRAM, the most restrictive PRAM. There are relatively few results on the EREW PRAM, and there is no deep understanding of the nature of the exclusive read restriction. The results that we present give only some insight into the problem.

In section 2, we have proved lower bounds for computing Boolean functions on CREW PRAMs. Clearly, these bounds hold for the EREW PRAM, too. In the case of the OR_n, we have obtained an exact bound, since Algorithm 2.1 is an EREW algorithm. If

memory cells of wordsize n can be used, then each Boolean function of n arguments can be computed by an EREW PRAM in $\lceil \log n \rceil + 1$ steps. No method is known to achieve the computation time matching the bound $\approx 0.72 \log n$, which is the lower bound for most Boolean functions of n arguments. (Recall that it is possible for CREW PRAMs with exponentially many processors.) For some functions, it is possible to achieve computation time $c \cdot \log n$ for a constant c, $c < 1$. We discuss such an algorithm for the PARITY$_n$ with running time $\approx 0.86 \log n$ [5, 7].

The algorithm computes PARITY for larger and larger groups of the input cells. When the computation terminates, there is exactly one group consisting of all input arguments. If a processor P knowing PARITY(A), for a subset of input arguments A, reads a cell that stores PARITY(B), for some subset B of the input arguments, and $A \cap B = \emptyset$, then processor P can add these values modulo 2 to compute PARITY$(A \cup B)$. If forming larger groups, with computing their parities, is confined to reading phases of the computation steps, then it is easy to see that the algorithm takes at least $\lceil \log n \rceil + 1$ steps. The key to a faster computation is to achieve forming larger groups during write phases. As for the OR$_n$, at first it seems to be impossible, since by writing, the old content of a cell is always overwritten. However, we can apply the following trick. Suppose that there are a processor P knowing a number $x \in \{0, 1\}$ and cells C_0, C_1, each storing a number $y \in \{0, 1\}$. Processor P writes symbol '$*$' into cell C_x. Then the cell C_{1-x} still stores y that is different from '$*$'. So if, after the writing, a cell C_i, for $i \in \{0, 1\}$, stores a symbol $j \neq *$, then we can conclude that $x = 1 - i$ and $y = j$. In that sense, the cell C_i codes both values x and y. This effect of appending information instead of overwriting combined with broadcasting techniques described in the next subsection (broadcasting is required since we need two copies of each cell before "appending" information) is the key to computing PARITY of 5 bits in two steps. Applying it as a subprocedure leads to an algorithm with computation time approximately $0.86 \log n$:

Algorithm 6.1 *For each n, there is an n-processor EREW PRAM that computes* PARITY$_n$ *in $2 + \lceil \log_5(n/2) \rceil$ steps.*

6.1 Algorithms and lower bounds for chosen problems

To get a deeper insight into EREW PRAMs, we consider three problems, for which tight upper and lower bounds are known.

6.1.1 Range searching

Snir [22] defines a range searching problem of size n: For an input consisting of $n + 1$ numbers $x_1, \ldots, x_n$ and y such that $x_1 < x_2 < \ldots < x_n$, find the index i such that $x_i < y \leq x_{i+1}$ (by definition, $x_0 = -\infty$ and $x_{n+1} = \infty$). There is an algorithm that solves the range searching problem of size n in $O(\sqrt{\log n})$ time [22]. Snir shows a matching lower bound:

Theorem 6.1 *An EREW PRAM that solves the range searching problem of size n makes at least $\sqrt{\log n}$ steps.*

For the above theorem, we assume that the input numbers $x_1, \ldots, x_n, y$ are quite arbitrary; more precisely, they can be chosen from a fairly large set. The size of this set depending on

n and the number of processors and cell of an EREW PRAM is chosen so that Ramsey's theorem can be applied.

6.1.2 Broadcasting

The most crucial difference between EREW and CREW PRAMs is that, in the case of an EREW PRAM, broadcasting information from one cell to many processors cannot be done by a concurrent-write operation and requires a large number of steps. Beame, Kik and Kutyłowski [2] show that making m copies of a single cell requires $\Theta(\log m)$ steps:

Theorem 6.2 *Let $m \in \mathbb{N}$.*

- *Generating m copies of a single memory cell takes at least $\approx 0.53 \log m$ steps on every EREW PRAM (the exact value is known and denoted by $\kappa(m)$).*
- *Assume that $k \in \mathbb{N}$ and a cell C may store only the numbers $1, 2, \ldots, k$. There is an EREW PRAM that makes m copies of cell C in $\kappa(m) + O(\log \log k)$ steps.*

First we sketch the proof of the lower bound. Suppose that a cell C may contain only 0 or 1. Then the content of C can be described by a single Boolean variable x. Since this is the only input to the algorithm, each processor state or content of a memory cell during broadcasting is represented by a Boolean formula in variable x. There are only 3 such satisfiable formulas: $1, x, \neg x$. Recall that if s_1, s_2 are different states of a processor (cell) after step t, and f_1, f_2 are formulas representing s_1, s_2, then $f_1 \wedge f_2$ cannot be satisfied. It follows that, at each moment of the computation, a processor (a cell) can reach either only one state, represented by 1, or exactly two states, represented by x and $\neg x$. In the later case, we get a processor (cell) knowing the value of x. Let p_t (c_t) be the number of processors (cells) knowing the value of x after step t. We show that $p_{t+1} \leq p_t + c_t$, and $c_{t+1} \leq c_t + 2p_{t+1}$, for each t. The first inequality is obvious. The new processors knowing the value of x at step $t+1$ are these that read the cells knowing the value of x. There are c_t such cells, hence $p_{t+1} \leq p_t + c_t$. During the write phase, only the processors knowing the value of x can generate new cells knowing the value of x. Each such processor may reach only two different states, in each state can write into at most one memory cell. Hence the number of cells that a processor may affect at one step is at most two. At this moment, there are p_{t+1} processors knowing the value of x, hence $c_{t+1} \leq c_t + 2p_{t+1}$.

It follows from the proved inequalities that $c_t \approx (3.73)^t$. Hence to make m copies at least $\approx 0.53 \log m$ steps are necessary.

In order to get a matching upper bound, we show that one processor P that knows the value of $x \in \{0, 1\}$ can pass this information into 2 cells at one step. Take cells C_i, for $i = 0, 1$, each C_i storing initially number i. If $x = 0$, then P writes 0 into C_1. If $x = 1$, then P writes 1 into C_0. It is easy to see that, after writing, both C_0 and C_1 store number x. This trick leads to the optimal algorithm making c_t copies of a cell storing $x \in \{0, 1\}$ in t steps, where $c_t \approx (3.73)^t$ is defined by:

$p_0 = 0, \quad c_0 = 1$;

$p_{i+1} = p_i + c_i, \quad c_{i+1} = 2p_{i+1} + c_i \quad$ for $i \geq 0$.

Hence to create m copies, approximately $0.53 \log m$ steps suffice. For $x \in \{0, 1, \ldots, k-1\}$, constructing a time optimal algorithm is also possible: First we create $\lceil \log k \rceil + 1$ copies of cell C. Applying a straightforward algorithm, it takes $O(\log \log k)$ steps. Then in one

step, we create the binary representation of x, each bit stored in a separate cell. Next, applying the optimal algorithm for broadcasting single bits, m copies of each bit of the binary representation of x are made. Thereby, m copies of the binary representation of x are created. In $O(\log\log k)$ steps, we decode x from each binary representation, getting m cells storing x. The total computation time is $\approx 0.53\log m + O(\log\log k)$. One can even reduce the number of processors (cells) to $3m$, if the above algorithm is carefully designed.

6.1.3 Merging sorted strings

We consider the following merging problem: Given two sorted strings X and Y of n numbers each. Compute the sorted string of length $2n$ consisting of the elements of X and Y, each element occurring as many times as in X and Y together. We consider two versions of this problem: for the first one, the numbers occurring inside the input strings are 0's and 1's, only. For the second version, these elements belong to a big but still reasonable set of numbers, for instance $\{1,\ldots,n\}$. It is easy to construct CREW PRAMs solving these merging problems in a constant time. As an example, we consider merging of bit strings. If for each bit in X there is a processor that inspects this bit and the bit following it, exactly one processor, detecting the change from 0 to 1 in X, will know the number of 0's in X. This processor writes this number into a fixed memory cell without a write conflict. The number of 0's in Y can be determined in the same way, and the sum of these two numbers can be computed and written into a fixed cell. Then $2n$ processors read from this cell, and each of them produces one output bit. Note that this algorithm cannot be converted to a fast EREW algorithm, because of the concurrent-read operation during the last step that requires $\Omega(\log n)$ simulation steps. In this section, we discuss the results of Hagerup and Kutyłowski [13] concerning complexity of merging on EREW PRAMs.

We start with a simple argument showing that the second problem requires $\Omega(\log n)$ steps. We reduce the broadcasting problem to merging. Let us consider the following strings of length n: $w = <1,2,\ldots,n>$, $v_1 = <n,n,\ldots,n>$, and $v_2 = <1,n,n,\ldots,n>$. Merging w with v_1 and w with v_2 give results that differ on positions $2,3,\ldots,n$. Hence, in order to broadcast x, for $x \in \{1,n\}$, it suffices to merge string w with the string $<x,n,\ldots,n>$. By examining any of the numbers on positions $2,3,\ldots,n$ in the output string, it is possible to determine x. Since broadcasting x into $n-1$ locations must take $\Omega(\log n)$ steps, merging these strings takes $\Omega(\log n)$ steps, too. One might expect that the same bound holds for the first merging problem. However:

Algorithm 6.2 *Two sorted bit strings of length n can be merged in $O(\log\log n)$ time by an EREW PRAM.*

The algorithm is too technical to be sketched here, and we confine ourselves to few hints. Consider a rectangular tableau with $\sqrt{n}$ rows and $2n/\sqrt{n}$ columns, with column-major order, i.e., the first $\sqrt{n}$ elements of the tableau are the elements of the first column, ordered from the top to bottom, the next $\sqrt{n}$ elements make the second column, etc.. The initial step is to store input strings X, Y in such a tableau, X followed by the reverse of Y. Then each row of the tableau is sorted recursively (it is possible since to sort a row we need only to merge the first half of the row with the reverse of the second half). It is

not difficult to see that this yields the correct output string with the exception of at most one column. This "critical" column is the only column that contains both 0's and 1's in the output string. The critical column can be approximately located at an early stage of the computation. Consider a row i. Before sorting the rows, one can easily find the last 0 in X and the first 0 in Y lying in row i. Hence the number of 0's in row i, denoted by z_i , can be easily determined. Then the critical column after sorting the rows is the column z_i or z_{i+1}. Exact locating the critical column is not possible (a "broadcasting" argument can be applied). So the algorithm has to correct the values in the critical column without knowing where the critical column is. The solution is to write correct values into a small area that *surrounds* the critical column. Generating these values and their addresses is the most difficult part of the algorithm that we are not able to expose here.

A close relative of the merging problem can be considered, which is called rank-merging: Given two sorted sequences X and Y of length n each, mark each input number with an element of the set $\{1, \ldots, 2n\}$, called its *rank*, such that distinct input numbers receive distinct ranks, and such that if some input value is smaller than another value, then its rank is also smaller (the ranks should represent a correctly sorted output sequence). Intuitively, the rank of an input number is its position in the output sequence. Adding the restriction that each element of X or Y should receive a larger rank than the element preceding it in X or Y, if any (i.e., the relative order of the elements in X and Y is to be preserved), we obtain the problem of *stable rank-merging*. Hagerup and Kutyłowski [13] show that stable rank-merging requires $\Omega(\log n)$ EREW steps for inputs of length n, whereas unstable rank-merging can be performed by an EREW PRAM that runs in $O(\log \log n)$ time for bit strings of length n.

6.2 Open problems for EREW PRAMs

The crucial difference between the CREW and the EREW PRAM is that in the case of the EREW PRAM information is processed locally, with the "neighborhoods" defined dynamically through read and write operations. It seems obvious that such "local" computations should be slower than CREW computations, where an instantaneous broadcasting of a message to all processors is possible through a concurrent-write operation. There are examples proving this true, but only for restricted domains or functions over large domains [11]. Fich and Wigderson [10] present Boolean functions that can be computed quickly by CREW PRAMs and that require a long time to solve on EROW PRAMs. (Recall that the EROW PRAM is an EREW PRAM for which every cell has its owner, the only processor that may write into it.) They consider Boolean decision tree evaluation problems $D_{m,h}$. An input for $D_{m,h}$ is a string determining queries placed in the binary tree of height h, each query named by a binary string of length m, and a string of 2^m bits defining the values of 2^m possible queries. A standard CREW PRAM procedure computes $D_{m,h}$ in $O(\log m + \log h)$ steps. Using probabilistic methods, Fich and Wigderson show a much higher lower bound for the EROW PRAM:

Theorem 6.3 *Every (probabilistic) EROW PRAM computing $D_{3T,6T^2}$ requires at least $T/2$ steps.*

Because of a complicated information flow in the case of the EREW PRAM, it is not known how to generalize this result for EREW PRAMs.

For the CREW PRAM, time complexity of a Boolean function can be determined (up to a constant factor) in terms of block sensitivity. The lower bound holds for the EREW case, but the tight upper bound shown by Nisan apply only to the CREW PRAMs. For the EREW PRAM, no upper bound except the trivial $\lceil \log n \rceil + 1$ is known. So for functions with block sensitivity $o(n)$, there is a large gap between known lower and upper bounds. To close this gap, presumably new complexity measures (more related to information flow) should be discovered.

Specially interesting seems investigating EREW PRAMs of a small wordsize. For such machines, the information transfer is severely restricted. Determining lower time bounds for this case would say something substantial about communication complexity of Boolean functions.

References

[1] P. Beame and J. Hastad. Optimal bounds for decision problems on the CRCW PRAM. *J. ACM*, 36 (1989), 643–670.

[2] P. Beame, M. Kik and M. Kutyłowski. Information broadcasting by Exclusive Read PRAMs. *Submitted*.

[3] S. J. Bellantoni. Parallel random access machines with bounded memory wordsize. *Inform. and Comput.*, 91 (1991), 259–273.

[4] S. Cook, C. Dwork, and R. Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM J. Comput.*, 15 (1986), 87–97.

[5] $*$ M. Dietzfelbinger, M. Kutyłowski and R. Reischuk. Exact time bounds for computing Boolean functions on PRAMs without simultaneous writes. In *Proc. 2nd ACM Symposium on Parallel Algorithms and Architectures* (Heraklion), Association for Computing Machinery, 1990, 125–135.

[6] $*$ M. Dietzfelbinger, M. Kutyłowski and R. Reischuk. Exact lower bounds for computing Boolean functions on CREW PRAMs. A journal version of the first part of [5], *submitted*.

[7] $*$ M. Dietzfelbinger, M. Kutyłowski and R. Reischuk. Realistic time-optimal algorithms for Boolean functions on Exclusive-Write PRAMs. A journal version of the second part of [5].

[8] F. E. Fich, R. Impagliazzo, B. Kapron and V. King. Limits on the power of parallel random access machines with weak forms of write conflict resolution. *Preliminary draft* , 1991.

[9] F. E. Fich, and P. Ragde. *Personal communication.*

*the papers supported by DFG-Schwerpunktprogramm "Datenstrukturen und effiziente Algorithmen" are marked with $*$

[10] F. E. Fich and A. Wigderson. Towards understanding exclusive write. In *Proc. 1st ACM Symposium on Parallel Algorithms and Architectures* (Santa Fe), Association for Computing Machinery, 1989, 76–82.

[11] E. Gafni, J. Naor and P. Ragde. On separating the EREW and CROW models. *To appear in Theoret. Comput. Sci.*.

[12] T. Hagerup. *Personal communication* .

[13] ∗ T. Hagerup and M. Kutyłowski. Fast merging on the EREW PRAM. *Submitted*.

[14] T. Hagerup and M. Nowak. Parallel retrieval of scattered information. In *Proc. International Congress on Automata Languages and Programming* (Stresa), European Association for Theoretical Computer Science, 1989, 439–450.

[15] T. Hagerup and T. Radzik. Every ROBUST CRCW PRAM can efficiently simulate a PRIORITY PRAM. In *Proc. 2nd ACM Symposium on Parallel Algorithms and Architectures* (Crete), Association for Computing Machinery, 1990, 125–135.

[16] M. Kowaluk and K. Loryś. *Personal communication* .

[17] M. Kutyłowski. Time complexity of Boolean functions on CREW PRAMs. *SIAM J. Comput.* 20 (1991).

[18] ∗ M. Kutyłowski. Lower time bounds for computing Boolean functions on ROBUST PRAMs. *Preliminary draft* , 1991.

[19] N. Nisan. CREW PRAMs and decision trees. In *Proc. 21st ACM Symposium on Theory of Computing* (Seattle), Association for Computing Machinery, 1989, pp. 327–335.

[20] I. Parberry and P. Y. Yan. Improved upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM J. Comput.*, 20 (1991), 88–99.

[21] M. Szegedy. Algebraic methods in lower bounds for computational models with limited communication. Ph. D. dissertation, University of Chicago, Chicago, Illinois, 1989.

[22] M. Snir. On parallel searching. *SIAM J. Comput.*, 14 (1985), 688–708.