Feasible Time-Optimal Algorithms for Boolean Functions on Exclusive-Write PRAMs*

Martin Dietzfelbinger†
Fachbereich Informatik
Universität Dortmund
D–44221 Dortmund
Germany

Mirosław Kutyłowski‡
Heinz Nixdorf Institut and
Fachbereich Mathematik-Informatik
Universität–Gesamthochschule–Paderborn
D–33095 Paderborn
Germany

Rüdiger Reischuk§
Institut für Theoretische Informatik
Medizinische Universität Lübeck
D–23560 Lübeck
Germany

December 1994

*A preliminary version of these results has been presented at the Second Annual ACM Symposium on Parallel Algorithms and Architectures, Crete, July 1990.
This extended and improved version will appear in SIAM Journal on Computing.

†Partially supported by DFG grant ME 872/1-4 and by DFG-Forschergruppe "Effiziente Nutzung paralleler Systeme, Teilprojekt 4". This author was affiliated with the Universität–GH–Paderborn while this paper was written.

‡Partially supported by the Alexander von Humboldt-Stiftung (while visiting TH Darmstadt), by program RP.I.09 by the Polish government, by DFG grant ME 872/1-4 (while visiting Paderborn), and by KBN grants 2 1197 91 01 and 8 S503 002 07. This author was affiliated with the University of Wrocław while this paper was written.

§Partially supported by the International Computer Science Institute, Berkeley, California. This author was affiliated with the Technische Hochschule Darmstadt while this paper was written.

Abstract

It was shown some years ago that the computation time for many important Boolean functions of n arguments on concurrent-read exclusive-write parallel random-access machines (CREW PRAMs) of unlimited size is at least $\varphi(n) \approx 0.72 \log_2 n$. On the other hand, it is known that every Boolean function of n arguments can be computed in $\varphi(n) + 1$ steps on a CREW PRAM with $n \cdot 2^{n-1}$ processors and memory cells. In the case of the OR of n bits, n processors and cells are sufficient. In this paper it is shown that for many important functions there are CREW PRAM algorithms that almost meet the lower bound in that they take $\varphi(n) + o(\log n)$ steps, but use only a small number of processors and memory cells (in most cases, n). In addition, the cells only have to store binary words of bounded length (in most cases, length 1). We call such algorithms "feasible". The functions concerned include: the PARITY function and, more generally, all symmetric functions; a large class of Boolean formulas; some functions over non-Boolean domains $\{0, \ldots, k-1\}$ for small k, in particular parallel prefix sums; addition of n-bit-numbers; sorting n/l binary numbers of length l. Further, it is shown that Boolean circuits with fan-in 2, depth d, and size s can be evaluated by CREW PRAMs with fewer than s processors in $\varphi(2^d) + o(d) \approx 0.72d + o(d)$ steps. For the exclusive-read exclusive-write model (EREW PRAM) a feasible algorithm is described that computes PARITY of n bits in $0.86 \log_2 n$ steps.

Key words. parallel random-access machine, exclusive-write, concurrent-read, exclusive-read, parallel time complexity, Boolean functions, Boolean formulas, Boolean circuits, symmetric functions, parallel prefix, parity, addition, sorting

AMS(MOS) subject classifications. 68Q10, 68Q05, 68Q25

1. Introduction

1.1 Motivation

The parallel random-access machine (PRAM) is a powerful machine model that is often used for the design of parallel algorithms. Several variants of this model have been studied, which differ in the rules that regulate concurrent access to memory cells in shared memory. In this paper, we concentrate on exclusive-write machines (CREW and EREW PRAMs), which do not allow concurrent write access to shared memory cells. CREW PRAMs allow concurrent reading of one cell, EREW PRAMs do not. (Precise definitions of the models used will be given in Section 2. For a survey of PRAM models and algorithms for such models see [15, 27].)

The time complexity of Boolean functions on CREW PRAMs is quite well understood: using certain parameters corresponding to structural properties of such functions ("block-critical complexity" or "degree", see below) it is possible to characterize their time complexity on CREW PRAMs up to a constant factor [23]. Methods were developed that allow proving lower bounds for the time complexity of many functions that are exact up to a small additive constant [9, 12, 17, 25].

In this context, both for upper and lower bounds, we use the standard definition of the "abstract CREW PRAM" from [9]. Each computation step consists of three phases, a READ, a COMPUTE, and a WRITE phase, which are executed synchronously by all processors. (For details see Section 2.) This machine model abstracts from the cost of internal computations of the processors; the bounds hold regardless of the number of processors and the wordsize of the common memory cells. Such a strong model is perfectly acceptable for lower bound proofs. However, also the general upper bounds that hold for Boolean functions on such machines are formulated with respect to the abstract PRAM. We consider two examples of such statements.

Fact **1.1** *Every Boolean function of n arguments can be computed in $\lceil \log n \rceil + 1$ steps*[†] *by an EREW PRAM with n processors and n memory cells of wordsize n.*

The algorithm behind this fact (in a binary tree fashion, collect the whole input in one processor, which then computes and writes the output bit) is of limited value for concrete functions since it requires that the PRAM has a wordsize of n, i.e., the common memory cells store numbers of binary length up to n.

Before describing the second example, we introduce a function that plays a central role in the exact upper and lower bounds for computing Boolean functions on abstract CREW PRAMs. Let

$$\varphi(n) \ := \ \min\{j \mid F_{2j+1} \geq n\} \ , \text{for } n \geq 1 \ ,$$

where F_i denotes the ith Fibonacci number, i.e., $F_0 = 0, F_1 = 1$, and $F_{i+2} = F_{i+1} + F_i$, for $i \in I\!N$. Note that from the well-known formula $F_i = (\Phi^i - \hat{\Phi}^i)/\sqrt{5}$ for $\Phi = \frac{1}{2}(1 + \sqrt{5})$ and $\hat{\Phi} = \frac{1}{2}(1 - \sqrt{5})$ it easily follows that for $b := \Phi^2 = \frac{1}{2}(3 + \sqrt{5})$ we have

$$\log_b n \ \leq \ \varphi(n) \ \leq \ \log_b n + 1.34 \ , \quad \text{for all } n \geq 1 \ . \tag{1.1}$$

[†]Throughout the paper, log stands for the logarithm with respect to base 2

Also note that $\log_b 2 \approx 0.72$, and hence $\varphi(n) \approx 0.72 \log n$.

Consider the function $\mathrm{OR}_n(x_1, \ldots, x_n) \ := \ x_1 \vee x_2 \vee \cdots \vee x_n$. It is known ([9]) that OR_n can be computed by an EREW PRAM with n processors and n memory cells of wordsize 1 in $\varphi(n)$ steps. Using this algorithm, it is easy to observe that all Boolean functions of n variables can be computed on CREW PRAMs almost as fast as OR_n ([9], see also [17]):

FACT **1.2** *Every Boolean function f of n arguments can be computed in $\varphi(n) + 1$ steps by a CREW PRAM with $n \cdot 2^{n-1}$ processors and $n \cdot 2^{n-1}$ memory cells of wordsize 1. Moreover, after step $\varphi(n) + 1$, there is a processor that knows all input bits.*

For many Boolean functions f this fact corresponds to an algorithm with an optimal running time [12]. But again, this algorithm (essentially, for every possible input vector $a = (a_1, \ldots, a_n) \in f^{-1}(1)$ there is a team of n processors that checks whether the actual input $(x_1, \ldots, x_n)$ equals a and if so writes the result 1 to the output cell) is practically worthless, because of the exponential number of processors required. Moreover, the algorithms to be carried out by the processors completely ignore any structure present in the function f: they simply represent the list of inputs in $f^{-1}(1)$. Algorithms of a similar kind are used in the general upper bound proofs of [23].

Our focus in this paper is on algorithms for computing specific functions on exclusive-write PRAMs that are time-optimal up to small *additive* terms and whose implementation is "feasible" in a sense discussed in the following. Specifically, we are interested in methods that do not use more than n processors and n common memory cells for computing functions of n arguments. For many interesting functions there are "critical inputs" (see [9] and Section 1.2), for which each input bit must be read by some processor; this implies that $\Omega(n/\log n)$ processors are necessary if logarithmic computation time is to be achieved.

How strongly the hardware size may influence the parallel time complexity is shown, for example, by the PARITY function:

$$\mathrm{PARITY}_n(x_1, \ldots, x_n) \ := \ x_1 \oplus x_2 \oplus \cdots \oplus x_n \ , \text{ for } n \geq 1 \ .$$

On a concurrent read *concurrent-write* (CRCW) PRAM with exponentially many processors and a common memory of exponential size this function can be computed in two steps. However, with only n processors the time complexity increases to $\Theta(\log n/\log\log n)$ [2]. A similar tradeoff holds with respect to the memory size. In general, we require that an n-processor machine has a common memory of size at least n. Otherwise, separate read-only input cells are necessary and the time complexity increases significantly due to the small communication bandwidth [32].

We may assume that the memory cells of a PRAM can store only binary numbers. Following [4], a PRAM is said to have wordsize w if the cells of its common memory can only hold binary words of length at most w. Although bounded wordsize may seem to be a reasonable requirement in combination with a processor of bound of n, it is a severe restriction, since for most Boolean functions of n arguments the computation time is at least $n/w - o(n/w)$ on PRAMs with n processors and wordsize w [4]. The algorithms for special Boolean functions that will be presented in this paper require only constant wordsize; in most cases wordsize 1 suffices. Note that in case the algorithms were implemented on a "concrete" PRAM with processors with local memories, this restriction on the wordsize would not apply to the memory cells or registers used by each processor in its local memory, which, e. g., have to hold addresses of common memory cells. These have to have wordsize at least logarithmic in the number of processors and global memory cells; this will also be sufficient for our algorithms.

Finally, we aim at simple programs for each processor. There seems to be no general agreement on what exactly this should mean, and we do not attempt to formalize this criterion; for a possible approach see for example [31]. Nonetheless, we feel that the PRAM algorithms presented in this paper fulfill this condition for any reasonable definition of "simple program".

The fundamental example of a feasible, time-optimal CREW algorithm is the method for computing OR_n in $\varphi(n)$ steps mentioned above [9], which even works on an EREW PRAM. It is feasible since only n processors and n memory cells are used, all cells have wordsize 1, and the processors execute a very simple program. At the first glance it may seem that at least $\log n$ steps are necessary for computing OR_n. The essential observation that makes it possible to achieve the speedup from $\log n$ to $\varphi(n)$ is that in certain situations a processor can transfer information into a common memory cell without destroying the information of that cell. This cannot be done by a direct WRITE to the cell since its prior contents would be overwritten; however, it can be achieved by *not writing.* This idea is exploited as follows in the algorithm for computing OR_n [9]: Assume that a processor P knows $y_1 \in \{0,1\}$ and a cell C stores a value $y_2 \in \{0,1\}$. By a single WRITE operation C can be set to the value of $y_1 \vee y_2$. Namely, if $y_1 = 0$, then P does not have to write, since $y_1 \vee y_2 = y_2$. If $y_1 = 1$, then P writes a 1 into C. Again, this gives the correct result, since $y_1 \vee y_2 = 1 \vee y_2 = 1$. Using this trick, in one computation step (consisting of a READ, a COMPUTE, and a WRITE phase) the processors of a CREW PRAM can increase the amount of information stored in common memory cells by more than a factor of 2. As a consequence, it is possible to compute OR_n in fewer than $\log n$ steps.

1.2 Lower and upper bounds

Essentially, two general methods are known for proving lower bounds for the time complexity of Boolean functions on CREW PRAMs. The first one is based on the concept of *critical complexity.* The critical complexity $c(f)$ of a function $f : \{0,1\}^n \to \{0,1\}$ is defined as the maximal number k for which there is an input $a = (a_1, a_2, \ldots, a_n)$ and a set $J \subseteq \{1, 2, \ldots, n\}$ of cardinality k such that

$$f(a) \neq f(a_1, \ldots, a_{i-1}, \neg a_i, a_{i+1}, \ldots, a_n) \quad , \quad \text{for all } i \in J \; .$$

A "critical input" a is one for which this condition is true with $J = \{1, \ldots n\}$. A lower bound for CREW PRAMs on the basis of $c(f)$ has been proved in [9] (improved in [25]):

FACT **1.3** *The time for a CREW PRAM to compute a function f is at least $\frac{1}{2} \log c(f)$, no matter how many processors and cells are used. Furthermore, the wordsize may be arbitrarily large, and the computational power of the processors may be unlimited.*

A generalization of the critical complexity, the so-called *block-critical complexity* (or "block sensitivity"), $bc(f)$, was considered in [23], where it was shown that the lower bound of Fact 1.3 can be improved to $\frac{1}{2} \log bc(f)$, and that the time complexity of f an abstract CREW PRAMs is $O(bc(f))$.

The second general lower bound method is based on an approach proposed in [17]. The idea is to consider the *degree* of a Boolean function when regarded as a polynomial over the integers. This useful complexity measure and variants thereof have applications far beyond complexity analysis of the CREW model, see for example [6, 24, 28, 29]. (The notion of "degree" of a Boolean function used in the lower bound proof for CRCW PRAMs of [2] is different.) Each Boolean function f can be represented by a polynomial over $x_1, \ldots, x_n$ with coefficients from $\mathbb{Z}$. This representation is unique if each x_i^d, for $d > 1$, $1 \le i \le n$, is reduced to x_i. The degree of the polynomial representing f is called the degree of f, denoted by $\deg(f)$. In [12], the authors have shown the following:

FACT **1.4** *At least $\varphi(\deg(f))$ steps are required for computing a Boolean function f on an arbitrarily large and powerful CREW PRAM.*

For the function OR_n this gives the lower time bound $\varphi(n)$, which, in view of the upper bound obtained in [9], is best possible. The lower bounds based on critical complexity and on degree complexity are not identical in the sense that there are functions f such that $\frac{1}{2}\log(bc(f))$ is smaller by a constant factor than $\varphi(\deg(f)) \approx 0.72\log(\deg(f))$ and vice versa. However, $\deg(f)$ and $bc(f)$ are polynomially related [12, 23, 29]; hence these two lower bounds differ from each other and from the CREW complexity of f at most by a constant factor.

There are Boolean functions that nontrivially depend on n arguments and still can be computed in $o(\log n)$ steps on a CREW PRAM. But for almost all functions of n arguments we have $c(f) \ge n-1$ [7], which implies a lower bound $\frac{1}{2}\log(n-1)$. One can even show that almost all functions of n arguments have degree n [12], hence they cannot be computed faster than in $\varphi(n)$ steps. On the other hand, we have the upper bound $\varphi(n)+1$ noted in Fact 1.2, which holds for all Boolean functions.

1.3 Results

In this paper, we will consider the following general problem.

> *Let a Boolean function f on n arguments with $\deg(f) = \Omega(n)$ be given. Design a feasible CREW PRAM algorithm that computes f in $\varphi(n) + o(\log n)$ steps, i. e., in almost optimal time.*

In general, such algorithms do no exist, by the results in [4]: For most Boolean functions of n arguments, if the number of processors is polynomially bounded in n and the wordsize is bounded by $o(n/\log n)$, then the CREW complexity becomes much larger than $\log n$. Nonetheless, we will construct feasible algorithms for many important and natural functions.

First, we consider a variant of the task just mentioned for EREW PRAMs, which seems to be substantially harder than for CREW PRAMs, due to the following observations. No lower bounds for EREW PRAMs are known that would not be valid for CREW PRAMs as well. On the other hand, most known CREW PRAM algorithms use the concurrent-read operation in a substantial way. Moreover, no fast simulation of the concurrent-read operation on EREW

PRAMs is possible [3], which makes it necessary to design efficient EREW algorithms by completely different techniques than CREW algorithms. There are some results elaborating on what EREW PRAMs can do less efficiently than CREW PRAMs [3, 14, 30], but still a deep understanding of the EREW PRAM model is missing. We will construct a feasible algorithm for the EREW PRAM that computes PARITY_n in approximately $0.86 \log n$ steps (Theorem 3.1). This is the second example—after the algorithm for OR_n—of an EREW algorithm with time complexity below $\log n$ for a function of degree n.

For the CREW PRAM model we will obtain the following results with respect to PARITY_n:

- PARITY_n can be computed in $\varphi(n)+1$ steps with $2^{O(\log^2 n)}$ processors and cells of wordsize 1, i. e., with significantly less than exponential hardware size (Theorem 4.1).
- PARITY_n can be computed in time $\varphi(n) + O(\sqrt{\log n})$ with n processors and n cells of wordsize 1 (Theorem 5.1).

Computing OR_n or PARITY_n amounts to evaluating a formula built with an associative operator over a 2-valued domain. We may generalize the results for these special functions to Boolean functions that are described by formulas or circuits:

- Any Boolean circuit of depth d and size s (with gates of fan-in 2) can be evaluated in $\varphi(2^d) + O(d/\log\log d) \approx 0.73d + o(d)$ steps by a CREW PRAM with $o(s)$ processors (Theorem 5.5); if the circuit is a formula (i. e., the gates have fan-out 1), then $2^d/\log d$ processors can evaluate it in time $\varphi(2^d) + O(d/\log d)$ (Theorem 5.6).

The results just mentioned can further be generalized to formulas $F(x_1, \ldots, x_n) = x_1 \otimes \cdots \otimes x_n$, where $\otimes$ is an associative binary operator over a k-valued domain, say, $\{0, 1, \ldots, k-1\}$, for some $k \geq 2$:

- Any such F can be computed in $\varphi(n) + 1$ steps on a CREW PRAM with $2^{O(\log^2 n \cdot \log k)}$ processors and cells of wordsize 1, with the obvious exception of the cells storing the input and the output (Theorem 6.1).
- Any such F can be computed in $\varphi(n) + O(\sqrt{\log n \cdot \log k})$ steps on a CREW PRAM with n processors and n cells of wordsize $\|k-1\|$, where $\|r\|$ denotes the number of digits in the binary representation of r, i. e., $\|r\| = \lceil \log(r+1) \rceil$ for integers $r > 0$ (Theorem 6.4).

The results concerning Boolean circuits and formulas may also be generalized to arbitrary circuits and formulas over k-valued domains (Theorems 6.5 and 6.6).

Further, we develop a method for computing in parallel all prefix products $x_1 \otimes \cdots \otimes x_i$ of $x_1 \otimes x_2 \otimes \cdots \otimes x_n$, for an arbitrary associative operator $\otimes$ over a k-valued domain. The complexity bounds are the same as those for computing the product only (Theorem 7.3). The parallel prefix operation has many applications; we consider just one of the most important ones:

- An n-processor CREW PRAM with a common memory of size n and wordsize 2 can add two binary numbers of length n in time $\varphi(n) + O(\sqrt{\log n})$.

In Section 8 we will present feasible algorithms with almost optimal running time for symmetric functions:

- Every symmetric Boolean function of n variables can be computed in $\varphi(n) + O(\log^{2/3} n \cdot \log\log n)$ steps by an n-processor CREW PRAM with n memory cells of wordsize 1 (Corollary 8.9).

The methods developed for symmetric functions can be used to show that also the problem of sorting bits or numbers can be solved by feasible algorithms in almost optimal time:

- n bits can be sorted in time $\varphi(n) + O(\log^{2/3} n \cdot \log\log n)$ by an n-processor CREW PRAM with n memory cells of wordsize 1 (Theorem 9.1).
- An $m^2 \cdot k$ processor CREW PRAM can sort m binary numbers of length k in time $\varphi(m \cdot k) + O(\log^{2/3} m \cdot \log\log m)$ using $m \cdot (m+1) \cdot k$ memory cells of wordsize 1 (Theorem 9.3).

Remark **1.5** Theorem 5.5 (mentioned above) provides a general way for obtaining a fast, feasible CREW PRAM algorithm for a Boolean function f of n variables, as follows:

(1) Design a circuit (with gates of fan-in 2) for f with depth d as small as possible and size $s = O(n)$;

(2) evaluate the circuit via the algorithm given in Theorem 5.5.

(Recall that if f is nondegenerate, i. e., f depends on all n variables, we must have $d \geq \log n$.) In some cases, it will even be possible to describe f by a formula of small depth over a 2-valued or a k-valued domain, for small k, and thus benefit from Theorem 5.6 or 6.6 (mentioned above). Note that step (1) has been carried out for many functions already, so results from the literature can be used. This approach will yield close to optimal time bounds ($\varphi(n) + o(\log n)$), with a small number of processors ($o(n)$) for some functions treated in this paper, like OR_n or PARITY_n or the addition of two binary numbers of n bits. However, in all these cases the additional $o(\log n)$-term is much smaller in the direct approach than in the algorithm resulting from the general method. For some other functions, e. g., the symmetric functions or the sorting function, no circuits are known whose simulation would yield a feasible algorithm with a running time of $\varphi(n) + o(\log n)$, since the best linear size circuits known have depth $(1 + \Omega(1)) \log n$. The most drastic example of the failure of this approach to constructing fast CREW PRAM algorithms is provided by the well-known storage access function

$$\mathrm{SA}_k \colon \{0,1\}^{k+2^k} \ni (y_{k-1}, \ldots, y_0, x_0, \ldots, x_{2^k-1}) \mapsto x_{(y_{k-1},\ldots,y_0)_2} \in \{0,1\} \ ,$$

where $(y_{k-1}, \ldots, y_0)_2 \in I\!N$ is the number with binary representation $y_{k-1} \ldots y_0$. The minimum depth of a circuit for this function is $k + \log k + \Theta(1)$ [34, p. 78], but there is a CREW PRAM algorithm for this function that uses 2^k processors, i. e., fewer than the number of variables, and has running time $\varphi(k) + O(1)$, i. e., only logarithmic in the depth of the circuit (use Lemma 8.8 below).

2. Preliminaries

We recall the definition of abstract CREW and EREW PRAMs (cf. [9]). A parallel random access machine consists of some number of processors $P_1, P_2, \ldots$ and common memory cells $C_1, C_2, \ldots$, which can be read from and written to by each of the processors. The computation proceeds in steps; each step consists of three phases. During the first phase a processor may read from a memory cell (the READ phase); during the second phase a processor changes its internal state according to the information read; during the third phase a processor may write into one memory cell (the WRITE phase). More precisely, we can describe the way a PRAM M works as follows. Let Q be the set of internal states of the processors of M and Σ the set of symbols that can be written into the memory cells. Associated with each processor P_i of M there are an initial state $q_i^0 \in Q$; a read-address function $\rho_i : Q \to I\!N$; a state transition function $\delta_i : Q \times (\Sigma \cup \{\$\}) \to Q$, for $\$ \notin \Sigma$; a write-address function $\tau_i : Q \to I\!N$; and a write-value function $\sigma_i : Q \to \Sigma$.

During a single step, if processor P_i is in state q, then it reads from cell C_j, where $j = \rho_i(q) \geq 1$; it does not read if $\rho_i(q) = 0$. If P_i reads a symbol $u \in \Sigma$, then the state of P_i changes to $q' = \delta_i(q, u)$. If P_i has decided not to read, then P_i changes its state to $q' = \delta_i(q, \$)$. During the third phase, P_i writes a symbol $v = \sigma_i(q')$ into cell $C_{j'}$, where $j' = \tau_i(q')$ if $\tau_i(q') > 0$, and does not write if $\tau_i(q') = 0$.

A PRAM M is a CREW (concurrent-read exclusive-write) PRAM if for no admissible initial set of values stored in the memory cells it happens that during some step in the computation of M two or more processors of M write into the same memory cell. In other words, M is a CREW PRAM if no write conflicts occur during a computation of M. A CREW PRAM M is an EREW (exclusive-read exclusive-write) PRAM if additionally two processors never *read* from the same cell during any step.

We say that a PRAM M computes a function f in T steps if the following holds. Initially, the arguments of f are stored in some fixed memory cells of M, each argument in a separate cell. After executing at most T steps the value of f on the given arguments is stored in one or several dedicated memory cells and all processors have stopped their computations. Here we are mainly interested in computing Boolean functions, that is, functions f of the form $f : \{0,1\}^n \to \{0,1\}^m$, for n, $m \in I\!N$. A PRAM is said to have wordsize $\leq w$ for some $w \geq 1$ if $|\Sigma| \leq 2^w$.

3. Fast Computation of PARITY on EREW PRAMs

This section deals with the EREW model. Our goal is to show that, like the OR, the PARITY of n bits can be computed in fewer than $\log n$ steps. The key to the method is a way for computing the PARITY of five bits in just two steps. Two technical tricks are necessary to achieve this goal. The first one is useful for CREW PRAMs, too, and will be exploited again in later sections. The second one is specific for EREW PRAMs and one of the basic elements of a time-optimal feasible broadcasting EREW algorithm presented in [3].

The first trick: If an algorithm wants to compute the PARITY of n bits in fewer than $\log n$ steps, it must in a single writing phase combine information known to a processor with some other information stored in a memory cell. By direct writing this is not possible since the old content of the memory cell would be overwritten. As in the fast OR-algorithm of [9] processors will transmit the information by *not writing.* The details are as follows. Let processors P_0 and P_1 know a Boolean value y_1 and cells C_0, C_1 store a Boolean value y_2. (We say a processor P_i *knows* a value y at the end of step t if this value is a function of the state q of processor P_i at the end of step t. One may imagine that part of the state of P_i is a register that explicitly contains y.) In order to leave information in some cell that is sufficient to determine $y_1 \oplus y_2$, the following instruction is executed in parallel for $i = 0, 1$ (see Figure 1):

- P_i writes $*$ into cell C_i if and only if $y_1 \neq i$.

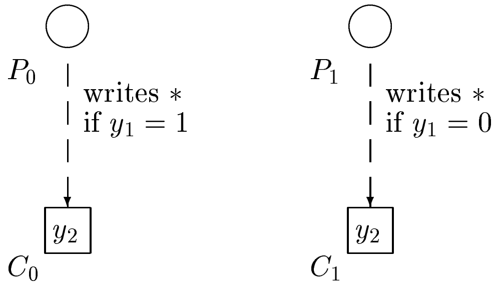

Figure 1: *The first trick: Combining information by not writing*

Afterwards, one memory cell contains $*$ while the other remains unchanged. Suppose that $y_1 = 0$. Then C_1 is set to $*$ and C_0 still contains y_2. If a processor reads C_0, it encounters a symbol z different from $*$, and can conclude that y_1 must be equal to 0. Hence, it can deduce that $y_1 \oplus y_2$ is equal to z. If $y_1 = 1$, then C_0 is set to $*$ while y_2 remains in C_1. If a processor reads z in C_1, it can deduce that $y_1 \oplus y_2 = 1 \oplus z$. In both cases, the processor that has read (C_0 or C_1, respectively) knows $y_1 \oplus y_2$.

The second trick: In order to use the first trick two different memory cells are needed that store the same value. We show how in one step a single processor can "write" a single value into two different cells (see Figure 2). Suppose that a processor P has to write $y \in \{0, 1\}$ into cells C_0, C_1. The cells C_0, C_1 are prepared in advance so that C_i contains i. If $y = 0$, then it suffices to change the contents of C_1 from 1 to 0, otherwise C_0 has to be corrected. Hence in each case it suffices that processor P writes y into the cell C_{1-y}. In the algorithm below the preparation of C_0 and C_1 will be done by free processors immediately before the writing of P occurs. Thus, no additional time is needed for the preprocessing.

THEOREM **3.1** *For each n, the function* PARITY_n *can be computed by an n-processor EREW PRAM with $2n$ memory cells of wordsize 1 in time*

$$2 + 2 \left\lceil \log_5 \tfrac{n}{2} \right\rceil \approx 0.86 \cdot \log n \ .$$

Proof. For the sake of simplicity we assume that $n = 2 \cdot 5^k$ for some $k \in I\!N$. In the first step of the EREW algorithm and the READ phase of its second step each input x_l, $1 \leq l \leq n$, is copied to an additional cell (say x_l from input cell C_l to C_{n+l}) and for each j, $1 \leq j \leq \frac{n}{2}$, two processors (say P_{2j-1} and P_{2j}) compute $\text{PARITY}(x_{2j-1}, x_{2j})$.

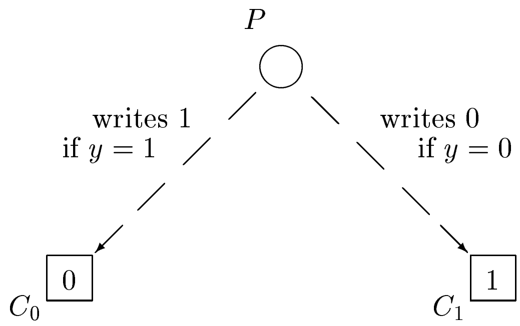

Figure 2: *The second trick: Writing two identical bits in one step*

After this preprocessing the main procedure starts. It consists of k stages, where each stage comprises 4 phases of alternating WRITEs and READs, starting with a WRITE. Thus a stage is made up of the last part of an EREW step (a WRITE), a complete step and the first part of another step (a READ).

The input for stage i for $i = 1, \ldots, k$, is described by a sequence of Boolean values $y_1^i, y_2^i, \ldots, y_{n_i}^i$, where $n_i = n/5^{i-1}$, such that

$$\text{PARITY}(x_1, ..., x_n) \;=\; \text{PARITY}(y_1^i, y_2^i, \ldots, y_{n_i}^i) \;.$$

The following properties are demanded at the beginning of each stage i:

- For each $1 \le l \le n_i$ there exist two cells that contain the value y_l^i .
- For every $1 \le j \le n_i/2$ two processors know $\text{PARITY}(y_{2j-1}^i, y_{2j}^i)$.

Assume that these properties hold at the beginning of stage i. Divide $y_1^i, y_2^i, \ldots, y_{n_i}^i$ into groups of ten elements each:

$$\{y_1^i, \ldots, y_{10}^i\},\ \{y_{11}^i, \ldots, y_{20}^i\},\ \ldots .$$

For each j, to get the cells storing $y_{2j-1}^{i+1}, y_{2j}^{i+1}$ and the processors knowing $\text{PARITY}(y_{2j-1}^{i+1}, y_{2j}^{i+1})$ the machine uses only the cells and processors associated with the jth group. Since the computation for each group is essentially the same, we describe only how y_1^{i+1}, y_2^{i+1} and $\text{PARITY}(y_1^{i+1}, y_2^{i+1})$ are computed. To simplify the exposition, processors and cells are named as follows. Let at the beginning of stage i

processors	P_1, P_2	know	$\text{PARITY}(y_1^i, y_2^i)$,
processors	P_3, P_4	know	$\text{PARITY}(y_3^i, y_4^i)$,
processors	P_7, P_8	know	$\text{PARITY}(y_7^i, y_8^i)$,
processors	P_9, P_{10}	know	$\text{PARITY}(y_9^i, y_{10}^i)$,
cells	C_5, C_5'	contain	y_5^i and
cells	C_6, C_6'	contain	y_6^i .

1st WRITE: During the first WRITE the information known to processors P_1, P_2 is combined with the information stored by the cells C_5, C_5' (and the information known by P_7, P_8 with that

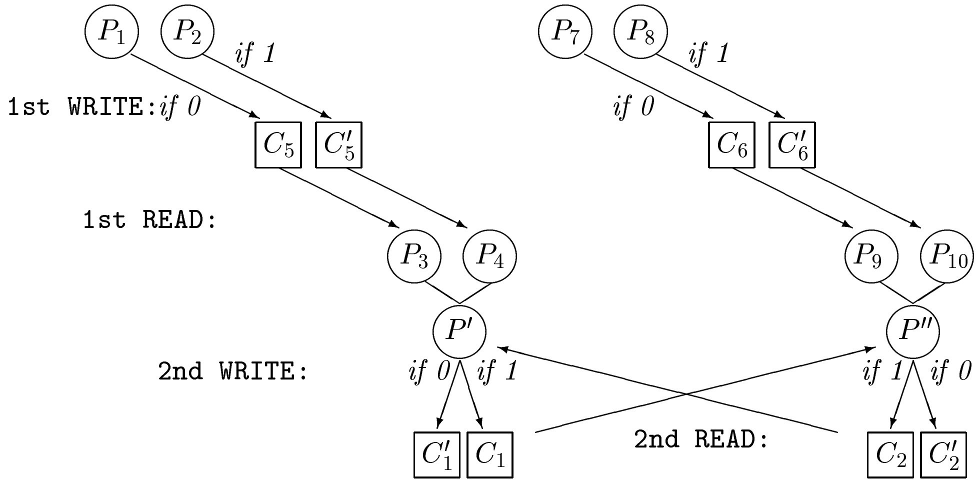

Figure 3: *Flow of information within a stage*

stored in C_6, C'_6), using the first trick described above:
- If $\text{PARITY}(y_1^i, y_2^i) = 1$, then P_1 writes $*$ into C_5.
- If $\text{PARITY}(y_1^i, y_2^i) = 0$, then P_2 writes $*$ into C'_5.
- If $\text{PARITY}(y_7^i, y_8^i) = 1$, then P_7 writes $*$ into C_6.
- If $\text{PARITY}(y_7^i, y_8^i) = 0$, then P_8 writes $*$ into C'_6.

1st READ: Since the other processors do not know which of the cells C_5, C'_5 contain the full information about $\text{PARITY}(y_1^i, y_2^i, y_5^i)$, we use two processors to read both cells in parallel.
- Processor P_3 reads C_5 and processor P_4 reads C'_5.

One of them encounters $*$ and terminates its work. The other one, call it P', using the information read and its own knowledge of $\text{PARITY}(y_3^i, y_4^i)$, determines

$$\text{PARITY}(y_1^i, y_2^i, y_3^i, y_4^i, y_5^i) = y_1^{i+1}.$$

- Processor P_9 reads C_6 and processor P_{10} reads cell C'_6.

Similarly, one of them halts while the other one, call it P'', computes

$$\text{PARITY}(y_6^i, y_7^i, y_8^i, y_9^i, y_{10}^i) = y_2^{i+1}.$$

2nd WRITE:
The purpose of this writing round is that P' writes y_1^{i+1} into some cells C_1 and C'_1 and P'' writes y_2^{i+1} into some cells C_2 and C'_2. For this, we apply the second trick described above. Thus, the cells C_1, C'_1, C_2, C'_2 must be prepared so that C_1 and C_2 contain 0 and C'_1 and C'_2 contain 1 (this can be done by processors P_3, P_4, P_9, P_{10} during the first WRITE). Then
- P' writes 1 into cell C_1 if $y_1^{i+1} = 1$ and writes 0 into C'_1 if $y_1^{i+1} = 0$.
- P'' writes 1 into cell C_2 if $y_2^{i+1} = 1$ and writes 0 into C'_2 if $y_2^{i+1} = 0$.

2nd READ:
- Processor P' reads cell C_2 and processor P'' reads cell C_1.

Now, knowing y_1^{i+1} and y_2^{i+1}, both can compute $\text{PARITY}(y_1^{i+1}, y_2^{i+1})$.

After performing stage i, there will be two cells containing y_1^{i+1} (cells C_1, C_1'), two cells containing y_2^{i+1} (cells C_2, C_2'), and two processors (P' and P'') knowing PARITY(y_1^{i+1}, y_2^{i+1}), as required.

Note that during this procedure concurrent READ or WRITE operations do not occur, as long as each pair of processors (P_1, P_2), (P_3, P_4), (P_7, P_8) and (P_9, P_{10}) agree on their specific role. Assuming that this is guaranteed for stage i, it can also be achieved for the next stage by declaring P' (the unique "survivor" of P_3, P_4) as first. Also, stages can easily be lined up, since the communication between stages is through fixed memory cells.

For $n = 2 \cdot 5^k$, the algorithm uses k stages to compute PARITY$(y_1^k, y_2^k) =$ PARITY$(x_1, \ldots, x_n)$. In an extra writing phase the first processor of the two processors knowing PARITY(y_1^k, y_2^k) writes the result into the output cell. The total number of steps is equal to

$$\tfrac{3}{2} + 2k + \tfrac{1}{2} \;=\; 2 + 2 \cdot \lceil \log_5 \tfrac{n}{2} \rceil \;\approx\; 0.86 \cdot \log n \ .$$

□

REMARK **3.2** The upper time bound for computing PARITY on EREW PRAMs presented above is not optimal. One can modify the algorithm to get a slightly smaller computation time. However, the construction is much more complicated, and therefore will be omitted.

4. A time-optimal CREW algorithm for PARITY with subexponentially many processors

Time bounds for computing the PARITY function on machines with a bounded number of processors have extensively been studied. For the most powerful PRAM model, the CRCW PRAM, in [2] an optimal lower bound of order $\log n / \log\log n$ is proved for the case where the number of processors is bounded by a polynomial. In [12] it has been shown that computing PARITY$_n$ on a CREW PRAM takes at least $\varphi(n)$ steps (and for some n even $\varphi(n) + 1$, see [18]) regardless of the number of processors. In this section we will prove that the time bound $\varphi(n) + 1$ can be achieved for PARITY$_n$ with much less than exponential hardware size. The algorithm described in this section is not feasible in the sense discussed in Section 1, because it uses too many processors and cells and hence too large addresses as well. However, in the next section the algorithm will be used as a component of a feasible algorithm for PARITY$_n$ that is almost time optimal.

THEOREM **4.1** *Let $n = F_{2t-1}$. Then* PARITY$_n$ *can be computed by a p-processor CREW PRAM M with m common memory cells of wordsize 1 in $t = 1 + \varphi(n)$ steps, where $p = n \cdot 2^{(t+1)t/2} \approx n \cdot 2^{(0.26 \log^2 n)}$ and $m = n \cdot (2^{t-1} + 1) \approx n^{1.72}$.*

Proof. The only property of PARITY$_n$ used in the construction below is that this function can be computed by an iteration of an associative binary operator. This makes it possible to generalize the construction for PARITY$_n$ to any associative operator (see Theorem 6.1), but here we will describe only the simple case of PARITY$_n$.

The CREW PRAM M constructed below combines the method used in [9] to compute the logical OR with the technique used by the fast EREW PRAM algorithm of Section 3. Since the algorithm is quite involved, we will describe it incrementally, each time giving more technical details. In order to achieve the computation time $\varphi(n) + 1$, during each WRITE the processors must combine their knowledge with the information stored in the memory cells. By direct writing, that is by overwriting, this is not possible; instead, we will use a variation of the "first trick" from the previous section.

There are n groups of $2^{(t+1)t/2}$ processors and n groups of 2^{t-1} memory cells. (Each group corresponds to a single processor or a cell from the algorithm of [9] for computing OR_n.) During the computation, each processor and each cell is either active or dead. (Dead cells are those that contain the symbol $*$.) Initially, all processors and cells are active. Once a cell or a processor becomes dead it remains in this state until the end of the computation. At any moment all active processors of a group have the same knowledge, and all active memory cells of a group code the same information about the input string. Which processors and cells are active depends on the current time step and on the input. A processor changes from the active to the dead state if it reads a dead cell; a cell changes to the dead state when $*$ is written into it. During the computation more and more processors and cells die, but for each group there is always a processor or a cell that is still active.

Each of the 2^{t-1} memory cells in a group has a *name*, which is a binary string of length $t-1$. The information carried by an active cell is not its content, but its name: the names of all active cells of a group agree in a prefix of a certain length, and this prefix codes all information the cells have about the input string. We briefly explain how this works. Let $\mathcal{C}$ be one of the groups of cells. During the first WRITE there is a group of processors that has to send a value $y_1 \in \{0,1\}$ to the memory cells of group $\mathcal{C}$.

- If $y_1 = 0$, then each cell of $\mathcal{C}$ with a name starting with 1 receives $*$, the cells with names starting with 0 remain unchanged.
- If $y_1 = 1$, then the roles are reversed: the cells in $\mathcal{C}$ with names starting with 0 are "killed", the other cells remain active.

During the second WRITE a different group of processors has to send a value $y_2 \in \{0,1\}$ to the cells of $\mathcal{C}$.

- If $y_2 = 0$, then all cells in $\mathcal{C}$ whose name has a 1 in the second position receive $*$.
- If $y_2 = 1$, then all cells in $\mathcal{C}$ whose name has a 0 in the second position receive $*$.

Obviously, the names of those cells in $\mathcal{C}$ that "survive" both WRITEs start with bits y_1, y_2. Similarly, during step s, all cells of group $\mathcal{C}$ are killed that have names whose sth bit differs from some y_s. In that way, after step s all active cells of $\mathcal{C}$ have names that agree in a prefix of length s. So if a cell of $\mathcal{C}$ with a name $u_1 u_2 \cdots u_{t-1}$ does not contain $*$ after step s, then it carries the information that $y_1 = u_1, y_2 = u_2, \ldots, y_s = u_s$, where $y_1, y_2, \ldots y_s$ are the bits representing the information transmitted to $\mathcal{C}$ by the processors. In that sense, at each step, information "written" by processors does not overwrite information carried by cells.

Why do we need so many processors in each group? During a single READ the active processors of one group read the memory cells of some other group. It would be best to read only the active cells; but as it is impossible to foresee which of the cells are active, the active processors are distributed evenly among the cells of the group and read all of them. Most of the processors die because they read dead cells, but always a small fraction survives. Because this fraction is getting smaller with each step, in each group the number of processors has to be much larger than the number of memory cells.

Now we describe the basic properties of the flow of information during the computation of M. For each $i \leq n$, during the first step M writes $*$ into each cell of group i whose name has the bit $\bar{x}_i$ in its first position. In this way x_i is coded by the memory cells of group i. Afterwards, the cell contents are not changed except that more and more cells die (receive a $*$). For each $i, j \leq n$, after step k,

- all active memory cells of group i code the value of $\mathrm{PARITY}(x_i, x_{i+1}, \ldots, x_{i+F_{2k}-1})$,
- all active processors of group j know the value of $\mathrm{PARITY}(x_j, x_{j+1}, \ldots, x_{j+F_{2k-1}-1})$

(compare with the algorithm in [9] for OR_n). It may happen that $i+F_{2k}-1 > n$ or $j+F_{2k-1}-1 > n$. So, for $l > n$, by x_l we mean $x_{l'}$ where $l' = l \bmod n$. Similarly, when we talk about a group i of processors (cells) and $i \notin \{1, \ldots, n\}$, then we mean a group i' for $i' = i \bmod n$.

Now let us describe what happens during step $k+1$. The active processors of group j read from the cells of group $j + F_{2k-1}$. A lot of processors encounter dead cells and terminate their work. Each of the active processors can deduce the value

$$\mathrm{PARITY}(x_{j+F_{2k-1}}, \ldots, x_{j+F_{2k-1}+F_{2k}-1}) = \mathrm{PARITY}(x_{j+F_{2k-1}}, \ldots, x_{j+F_{2k+1}-1})$$

from the address of the cell it reads, hence it can compute

$$\begin{aligned} & \mathrm{PARITY}(x_j, \ldots, x_{j+F_{2k-1}-1}) \oplus \mathrm{PARITY}(x_{j+F_{2k-1}}, \ldots, x_{j+F_{2k+1}-1}) \\ &= \mathrm{PARITY}(x_j, x_{j+1}, \ldots, x_{j+F_{2k+1}-1}) \\ &= \mathrm{PARITY}(x_j, x_{j+1}, \ldots, x_{j+F_{2(k+1)-1}-1}), \end{aligned}$$

which is the value the processors of group j have to know after step $k+1$. During the WRITE phase of step $k+1$ the active processors of group j write into some cells of group $j - F_{2k}$. The way of writing depends on the value

$$s = \mathrm{PARITY}(x_j, x_{j+1}, \ldots, x_{j+F_{2k+1}-1})$$

known to the processors. The symbol $*$ is written into all cells of group $j - F_{2k}$ with a name whose $(k+1)$st bit differs from s. Recall that the first k bits of the name of each cell that are still active before step $k+1$ code

$$\begin{aligned} & \mathrm{PARITY}(x_{j-F_{2k}}, x_{j-F_{2k}+1}, \ldots, x_{j-F_{2k}+F_{2k}-1}) \\ &= \mathrm{PARITY}(x_{j-F_{2k}}, x_{j-F_{2k}+1}, \ldots, x_{j-1}) \ . \end{aligned}$$

On the other hand, bit $k+1$ of the name of the cells that are active after step $k+1$ is equal to s, hence each such cell now codes

$$\begin{aligned} & \mathrm{PARITY}(x_{j-F_{2k}}, x_{j-F_{2k}+1}, \ldots, x_{j-1}) \oplus s \\ &= \mathrm{PARITY}(x_{j-F_{2k}}, \ldots, x_{j-1}) \oplus \mathrm{PARITY}(x_j, \ldots, x_{j+F_{2k+1}-1}) \\ &= \mathrm{PARITY}(x_{j-F_{2k}}, \ldots, x_{(j-F_{2k})+(F_{2k}+F_{2k+1}-1)}) \\ &= \mathrm{PARITY}(x_{j-F_{2k}}, \ldots, x_{(j-F_{2k})+F_{2(k+1)}-1}) \ . \end{aligned}$$

Note that this is the value that must be coded by the active cells of group $j - F_{2k}$ after step $k+1$. After step $t-1$, the remaining active processors of the first group know

$$\mathrm{PARITY}(x_1, x_2, \ldots, x_{F_{2t-3}}) \ .$$

They try to determine the value

$$\mathrm{PARITY}(x_{F_{2t-3}+1}, \ldots, x_{F_{2t-3}+F_{2t-2}}) \ = \ \mathrm{PARITY}(x_{F_{2t-3}+1}, \ldots, x_{F_{2t-1}}) \ ,$$

by reading from the remaining active cell of group $F_{2t-3}+1$. One of them survives the reading and computes

$$\begin{aligned} &\mathrm{PARITY}(x_1, x_2, \ldots, x_{F_{2t-3}}) \oplus \mathrm{PARITY}(x_{F_{2t-3}+1}, \ldots, x_{F_{2t-1}}) \\ &= \ \mathrm{PARITY}(x_1, x_2, \ldots, x_{F_{2t-1}}) \ , \end{aligned}$$

which it then writes into the output cell. Since $F_{2t-1} = n$, this is the correct output.

In order to finish the description of the algorithm we need to solve two problems: how to assign the processors to the cells for reading and how to choose the processors for writing to avoid a write conflict. There are $2^{(t+1)t/2}$ processors in each group named by binary strings of length $(t+1)t/2$. Let $s(0) = 0$ and for $k > 0$ define $s(k) = \sum_{l=1}^{k} l$. The memory cells and processors are assigned for reading and writing in such a way that after step k:

- in each group, the first k bits of the names of the active cells are identical, the remaining bits are arbitrary,
- in each group, the first $s(k-1)$ bits of the names of the active processors are identical, the remaining bits are arbitrary.

We show how the cells and processors are assigned for reading and writing during step $k+1$ to preserve these properties. During step $k+1$ each active processor of group j reads that memory cell of group $j + F_{2k-1}$ whose name agrees with the name of the processor from position $s(k-1)+1$ through $s(k-1)+(t-1)$. After step k the names of the active memory cells of group $j + F_{2k-1}$ have identical prefixes of length k. Hence after the READ operation the names of active processors in group j have the same prefix π_{k+1} of length $s(k-1)+k = s(k)$; the remaining bits of their names may be arbitrary.

Now we must select processors for the WRITE operation. The processors of group j have to write into cells of group $j - F_{2k}$. For a cell with a name $u_1 u_2 \cdots u_{t-1}$ we select the processor with the name $\pi_{k+1} u_1 u_2 \cdots u_{t-1} 00 \cdots 0$ (this is the unique active processor that has a name with suffix $u_1 u_2 \cdots u_{t-1} 00 \cdots 0$). This processor sends the symbol $*$ into the chosen cell if and only if $u_{k+1} \neq \mathrm{PARITY}(x_j, x_{j+1}, \ldots, x_{j+F_{2k+1}-1})$. Note that after this WRITE operation the names of the active cells of group $j - F_{2k}$ have the same bit $k+1$, so together all bits from 1 through $k+1$ are fixed.

After the last READ, for the names of active processors, a prefix of length $s(t) = (t+1)t/2$ is fixed, that is, a complete name. So exactly one active processor remains in each group. The active processor of the first group writes the result into the output cell. □

Remark **4.2** Simple but tedious modifications in the above algorithm make it possible to remove the assumption that n is a Fibonacci number from Theorem 4.1.

REMARK **4.3** Let $\Phi(y) = (\varphi(y)+2)\cdot(\varphi(y)+1)/2$ (hence $p = p(n) := n \cdot 2^{\Phi(n)}$ in Theorem 4.1). Knowing that $\log_b x \le \varphi(x) \le \log_b x + 1.34$, for $b = \frac{1}{2}(3+\sqrt{5})$ (eq. (1.1)), one can easily derive that $\Phi(n) \ge 0.25\log^2 n$, for all n, and that $\Phi(n) \approx 0.26\log^2 n$, for sufficiently large n. Already for $n \ge 16$ we have $\Phi(n) \le 4\log^2 n$. Hence $n \cdot 2^{0.25\log^2 n} \le p(n) \le n \cdot 2^{4\log^2 n}$, for $n \ge 16$.

5. Fast formula and circuit evaluation by CREW PRAMs with a linear number of processors

In the previous section we proved that the function PARITY_n can be computed in time $\varphi(n)+1$ with subexponentially many processors. In this section we show that PARITY_n can also be computed with a linear number of processors, while the computation time can be kept close to $\varphi(n)$. In a second step, the proof of this result is varied to obtain a method for evaluating Boolean circuits of depth d in $\varphi(2^d) + o(d) = 0.72...d + o(d)$ steps on CREW PRAMS with a linear number of processors.

THEOREM **5.1** *There is an n-processor CREW PRAM M with n cells of wordsize 1 that computes PARITY_n in $\varphi(n) + O(\sqrt{\log n})$ steps.*

Proof. We may assume that $n = 2^d$ for some integer d. The computation of M on input $x_1, \ldots, x_n$ proceeds in stages $i = 1, \ldots, m$, where m is determined below. The input for stage i are n_i binary values $y_1^i, y_2^i, \ldots, y_{n_i}^i$ stored in n_i fixed cells, so that the invariant

$$\mathrm{PARITY}(x_1, \ldots, x_n) \;=\; \mathrm{PARITY}(y_1^i, y_2^i, \ldots, y_{n_i}^i) \qquad (\mathcal{I}_i)$$

is maintained. A preprocessing phase in which groups of 8 processors each compute the PARITY of 8 bits in 4 steps (in the obvious binary tree fashion) allows us to assume that we can start with $n_1 = n/8 = 2^{d-3}$ values $y_1^1, \ldots, y_{n_1}^1$. The output of stage i, $1 \le i < m$, is the input sequence $y_1^{i+1}, y_2^{i+1}, \ldots, y_{n_{i+1}}^{i+1}$ for the next stage; the output of stage m is a single bit $y_1^{m+1} = \mathrm{PARITY}(x_1, \ldots, x_n)$, the desired result. Now we describe stage i, $i \ge 1$. We split the values $y_1^i, y_2^i, \ldots, y_{n_i}^i$ into disjoint groups $y_{js+1}^i, \ldots, y_{(j+1)s}^i$ of equal size s and compute PARITY within each group by the optimal algorithm of Theorem 4.1, which takes $\varphi(s)+1$ steps and needs $s \cdot (2^{\varphi(s)}+1) \le s \cdot 2^{\Phi(s)}$ cells and $s \cdot 2^{\Phi(s)}$ processors for each group, where $\Phi(s) = (\varphi(s)+2)\cdot(\varphi(s)+1)/2$ (cf. Remark 4.3). The $n_{i+1} := n_i/s$ output bits of the groups are $y_1^{i+1}, y_2^{i+1}, \ldots, y_{n_{i+1}}^{i+1}$. As n processors are available for a total of n_i bits, we can use sn/n_i processors for each group. If we define $w_i := \log(n/n_i)$, i.e., $n_i = 2^{d-w_i}$, then $s \cdot 2^{w_i}$ processors are available for each group. Thus, any s with $s \cdot 2^{\Phi(s)} \le s \cdot 2^{w_i}$, or $\Phi(s) \le w_i$, is suitable. Since it is convenient to have groups of size a power of 2, we let u_i be the maximal integer u such that $\Phi(2^u) \le w_i$, and define the group size s_i for stage i by $s_i := 2^{u_i}$. (Note that $w_i \ge w_1 = 3$, since $n_i \le n_1 = n/8$, and $\Phi(2^1) = 3$, by inspection; hence $u_i \ge 1$, for $i \ge 1$.) The only exception to this rule occurs in the last stage: Let $m := \min\{i \mid 2^{u_i} \ge n_i\}$; i.e., m is the minimal i with $w_i + u_i \ge d$. In stage m we form one group of size $s_i := n_i$ and use $n_i \cdot 2^{\Phi(2^{n_i})} \le 2^{d-w_i} \cdot 2^{w_i} = 2^d = n$ processors to compute PARITY of all $s_i := n_i$ bits left.

The correctness of the algorithm is obvious, as is the fact that only n processors and memory cells are used. We analyze the computation time. Note first that the equalities $s_i = n_i/n_{i+1}$, for $1 \le i < m$, and $s_m = n_m$ imply $\prod_{i=1}^m s_i = n_1 < n$. Using the inequalities $\log_b(z) \le \varphi(z) \le$

$\log_b(z) + 1.34$, valid for all z, by eq. (1.1), we may estimate the total number T of steps as follows:

$$T = \sum_{i=1}^{m}(\varphi(s_i)+1) \le \sum_{i=1}^{m}(\log_b(s_i)+2.34) = \log_b\left(\prod_{i=1}^{m} s_i\right) + 2.34m$$
$$\le \ \log_b(n) + 2.34m \le \varphi(n) + 2.34m\,.$$

In order to prove the theorem, it remains to estimate m. We may conclude from the equality $2^{d-w_{i+1}} = n_{i+1} = n_i/s_i = 2^{d-w_i-u_i}$ that $w_{i+1} = w_i + u_i$, for $1 \le i < m$. This implies $w_i = w_1 + \sum_{j=1}^{i-1} u_j$, for $1 \le i \le m$. The number m is characterized as the smallest i that satisfies $w_i + u_i \ge d$; thus, $m = \min\{i \mid w_1 + \sum_{j=1}^{i} u_j \ge d\}$. Now the desired estimate $m = O(\sqrt{\log n}\,)$ is given by the following lemma (choose parameters $A = 2$, $q = 0$, $\eta = 0$, and $c = 1$). □

For later use, we formulate and prove the technical lemma missing in the previous proof in a slightly more general way than needed here.

LEMMA **5.2** *Let integers $A \ge 2$, $q \ge 0$ and some constant $\eta \in \{0,1\}$ be fixed, let $d \ge 1$ be an integer, and $c \in [1,d]$ be arbitrary. If the integer sequence v_i, $i \ge 1$, is defined recursively by*

$$v_i = \max\{v \mid \eta\cdot v + c\cdot v^q\cdot\Phi(A^v) \le \eta + c\Phi(A) + \sum_{j=1}^{i-1} v_j\}\ ,\ \textit{for } i \ge 1\ ,$$

and $m = m(c,d)$ is defined by $m = \min\{i \mid \eta + c\Phi(A) + \sum_{j=1}^{i} v_j \ge d\}$, then

$$m = O((d^{q+1}\cdot c)^{1/(q+2)}).$$

(The constant factor in this bound depends on A and q.)

Proof. Obviously, $v_1 = 1$ and v_i, $i \ge 1$, is a nondecreasing sequence. For $l \ge 1$, let i_l denote the largest i such that $v_i \le l$ (the fact that $v_i \ge 1$ implies that i_l is a well-defined integer). Further, let $i_0 = 0$. Let $S_l = i_l - i_{l-1}$, the number of indices i with $v_i = l$. Obviously,

$$v_1 + \cdots + v_{i_l} = S_1 + 2S_2 + \cdots + lS_l\ ,\ \text{for } l \ge 1\ .$$

Let

$$l_0 = \min\{l \mid \eta + c\Phi(A) + S_1 + 2S_2 + \cdots + lS_l \ge d\}\,.$$

It is immediate from the definitions that $m \le \sum_{l=1}^{l_0} S_l$. The following two claims are sufficient to prove the lemma.

Claim 1. $S_l = O(c\cdot l^q)$, for $l \ge 1$.

Claim 2. $l_0 = O((d/c)^{1/(q+2)})$.

Namely, using the two claims, we have:

$$m \le \sum_{l=1}^{l_0} S_l = \sum_{l=1}^{l_0} O(c\cdot l^q) = O(c\cdot l_0^{q+1}) = O\left(c\cdot\left(\frac{d}{c}\right)^{(q+1)/(q+2)}\right) = O((d^{q+1}\cdot c)^{1/(q+2)})\,,$$

as desired.

Proof of Claim 1: Fix $l \geq 1$, and assume $S_l \geq 1$. (If $S_l = 0$, there is nothing to show.) From the definitions we have that

$$\eta \cdot l + c \cdot l^q \cdot \Phi(A^l) \leq \eta + c\Phi(A) + S_1 + 2S_2 + \cdots + (l-1)S_{l-1}$$

and

$$\eta + c\Phi(A) + S_1 + 2S_2 + \cdots + (l-1)S_{l-1} + l(S_l - 1) < \eta \cdot (l+1) + c \cdot (l+1)^q \cdot \Phi(A^{l+1}).$$

Adding these inequalities yields $l(S_l - 1) < \eta + c \cdot \Big((l+1)^q \cdot \Phi(A^{l+1}) - l^q \cdot \Phi(A^l)\Big)$, or

$$S_l \leq 1 + \frac{\eta}{l} + c \cdot \frac{1}{l} \cdot \Big((l+1)^q \cdot \Phi(A^{l+1}) - l^q \cdot \Phi(A^l)\Big)\,.$$

Since $1 + \eta/l \leq 2$, for proving Claim 1 it suffices to show that

$$(l+1)^q \cdot \Phi(A^{l+1}) - l^q \cdot \Phi(A^l) = O(l^{q+1})\,. \tag{5.1}$$

Recall that for $u \geq 1$ we have $\Phi(A^u) = (\varphi(A^u) + 1)(\varphi(A^u) + 2)/2$ and (cf. Remark 1.1)

$$\beta u = \log_b A^u \leq \varphi(A^u) \leq \log_b A^u + 1.34 = \beta u + 1.34\,,$$

where $\beta = \log_b A = \log(A)/\log((3+\sqrt{5})/2)$. Thus, we may estimate:

$$\begin{aligned}
&2 \cdot \Big((l+1)^q \cdot \Phi(A^{l+1}) - l^q \cdot \Phi(A^l)\Big) \\
\leq\;& (l+1)^q(\beta(l+1) + 2.34)(\beta(l+1) + 3.34) - l^q(\beta l + 1)(\beta l + 2) \\
=\;& \beta^2(l+1)^{q+2} - \beta^2 l^{q+2} + O(l^{q+1}) \\
=\;& O(l^{q+1})\,.
\end{aligned}$$

This proves (5.1) and Claim 1.

Proof of Claim 2: The definition of l_0 resp. $l_0 - 1$ implies that

$$\eta \cdot l_0 + c \cdot {l_0}^q \cdot \Phi(A^{l_0}) \leq \eta + c\Phi(A) + \sum_{j=1}^{i_{l_0-1}} v_j < d\,.$$

Substituting the inequality $\Phi(A^{l_0}) > \varphi(A^{l_0})^2/2 \geq \beta^2 l_0^2/2$, for $\beta = \log_b A$, we conclude that $c \cdot {l_0}^q \cdot \beta^2 l_0^2/2 < d$, which implies $l_0 = O((d/c)^{1/(q+2)})$. Thus, Claim 2 and Lemma 5.2 are proved. $\square$

Computing PARITY_n with $n = 2^d$ may be viewed as the problem of evaluating a certain Boolean formula that has the form of a complete binary tree of depth d with all operators equal to $\oplus$. A more general question is how fast Boolean formulas built from arbitrary binary operators may be evaluated or, even more generally, how fast Boolean circuits of depth d with gates of fan-in 2 may be evaluated. Of course, the best one can hope for is $\varphi(2^d) \approx 0.72d$ steps, since for example OR_n for $n = 2^d$ variables requires $\varphi(n)$ steps and has a circuit of depth d. In the following we show that circuits of depth d can indeed be evaluated in $\varphi(2^d) + o(d)$ steps on CREW PRAMs with quite small hardware expenditure. Subsequently we shall see that the special case of Boolean formulas (all gates have fan-out 1) allows a further reduction of the additive term $o(d)$.

For Boolean circuits, we use the standard notation as introduced e. g. in [34, p. 9]. We assume that all gates that have fan-in 1 or 2; there are no restrictions on the types of gates or on the fan-out. The depth of the circuit (i. e., the length of the longest path from an input to an output gate) is denoted by d, its size (i. e., the number of gates, not counting the inputs) by s. In order to compute functions with several outputs, some of the gates are marked as output gates. Since our algorithms will determine the values at *all* gates, the positions of the output gates are irrelevant.

We will need to subdivide the gates of a circuit into *levels* of a certain size (the *width*), in the following (slightly unusual) sense. We say that a circuit C *can be arranged in λ levels of width up to w* if the set of gates of C can be partitioned into *levels* $L_1, \ldots, L_\lambda$ with $|L_l| \leq w$ for $1 \leq l \leq \lambda$ such that a wire may only connect an output of a gate on level L_i with an input of a gate on level L_j if $i < j$. The n inputs for C form a separate level L_0, which is not subject to the width condition.

Trivially, every circuit C of depth d and size s can be arranged in d levels of width up to s. Using a simple variant of Brent's scheduling principle [5], we obtain arrangements with smaller width.

LEMMA **5.3** *Let C be a Boolean circuit of size s and depth d, and let $w \geq 1$ be arbitrary. Then C can be arranged in $d + \lfloor s/w \rfloor$ levels of width up to w.*

Proof. As noted above, C can be arranged in d levels $L_1, \ldots, L_d$ of width up to s. For $1 \leq i \leq d$, level L_i is further subdivided into $\lceil |L_i|/w \rceil$ sublevels: $\lfloor |L_i|/w \rfloor$ of these consist of w gates and at most one consists of fewer than w gates. An order for the sublevels within one level is fixed arbitrarily. Clearly, overall there are at most $\lfloor s/w \rfloor$ sublevels with exactly w gates and at most d sublevels with fewer than w gates. $\square$

The following technical lemma is the basis for our fast evaluation results for circuits and formulas.

LEMMA **5.4** *Let C be a Boolean circuit of size s that can be arranged in λ levels of width up to w. Let $\nu \geq 1$ be arbitrary. Then there is a CREW PRAM with $w \cdot 2^{2^\nu + \nu}$ processors and $s + w \cdot 2^{2^\nu + \nu}$ memory cells of wordsize 1 that for arbitrary inputs for C computes the values of all s gates of C in*

$$\log_b(2^\lambda) + O\left(\frac{\lambda}{\nu}\right) = 0.72... \cdot \lambda + O\left(\frac{\lambda}{\nu}\right)$$

steps.

Proof. The computation proceeds in stages $t = 1, 2, \ldots, \lceil \lambda/\nu \rceil$. For simplicity, we assume that ν divides λ; to cover the general case, only slight changes are necessary. During stage t, the gates in levels L_l for $(t-1)\nu + 1 \leq l \leq t\nu$ are evaluated simultaneously by groups of processors working independently. The values of the gates are stored permanently in s designated cells. Consider one gate g at level L_l. By the fan-in restriction and the fact that wires only run from lower-numbered to higher-numbered levels we know that the value of g is a function of the values of at most $2^{l-(t-1)\nu}$ gates in levels L_0 (the inputs), $L_1, \ldots, L_{l-(t-1)\nu}$, which are available at the beginning of stage t. By Fact 1.2, the value of g can be obtained in $\varphi(2^{l-(t-1)\nu}) + 1 \leq \varphi(2^\nu) + 1$ steps by

$$2^{l-(t-1)\nu + 2^{l-(t-1)\nu} - 1} \leq 2^{2^\nu + l - (t-1)\nu - 1}$$

processors, using the same number of memory cells. Summing over the (up to w) gates on level L_l and summing over $(t-1)\nu+1 \le l \le t\nu$, we see that stage t is finished in $\varphi(2^\nu)+1$ steps and requires

$$w \cdot \sum_{r=1}^{\nu} 2^{2^\nu+r-1} < w \cdot 2^{2^\nu+\nu}$$

processors and memory cells, besides the memory cells for (permanently) storing the newly computed values of the gates in L_l, $(t-1)\nu+1 \le l \le t\nu$. Since the same processors and memory cells can be used in all stages, the claimed bounds for these resources are proved. It remains to estimate the running time. Using eq. (1.1), we see that the total number of steps made in all stages is bounded by

$$\frac{\lambda}{\nu} \cdot (\varphi(2^\nu)+1) \le \frac{\lambda}{\nu} \cdot (\log_b(2^\nu)+2.34) = \log_b(2^\lambda) + O\left(\frac{\lambda}{\nu}\right) .$$

□

It is now only a matter of adjusting the parameters w and ν to obtain a CREW PRAM with fewer than s processors that can evaluate a circuit C of size s and depth d in time $0.72\ldots \cdot d + o(d)$.

THEOREM **5.5** *Let $f : \{0,1\}^n \to \{0,1\}^m$ be a Boolean function that is computed by a circuit C of depth d and size s.*

(a) *Let $\nu \ge 1$ and p be arbitrary. Then f can be computed by a CREW PRAM with p processors and $p+s$ memory cells of wordsize 1 in $\varphi(2^d) + O\left(2^{2^\nu+\nu} \cdot s/p\right) + O\left(d/\nu\right)$ steps.*

(b) *If p is such that* $\log\log(pd/s) \ge 3$, *then there is a CREW PRAM with p processors and $p+s$ memory cells of wordsize 1 that computes f in $\varphi(2^d) + O(d/\log\log(pd/s))$ steps.*

(c) *Assume* $\log\log(pd/s) \ge 3$. *If $p \ge s/\sqrt{d}$, the running time in* (b) *is $\varphi(2^d) + O(d/\log\log d)$; if $p \ge s/(d/\log d)$, it is $\varphi(2^d) + O(d/\log\log\log d)$.*

Proof. (a) If $2^{2^\nu+\nu} > p/2$, there is nothing to show, since then C can be evaluated in $O(s) = O(p \cdot (s/p)) = O(2^{2^\nu+\nu} \cdot (s/p))$ steps by one processor. Thus, assume $p \ge 2^{2^\nu+\nu+1}$. Let $w = \lfloor p/2^{2^\nu+\nu} \rfloor \ge 2$, and apply Lemma 5.3 to see that C can be arranged in $\lambda = d + \lfloor s/w \rfloor$ levels of width w. By Lemma 5.4, C can be evaluated by $w \cdot 2^{2^\nu+\nu} \le p$ processors in time

$$\begin{aligned}
\log_b(2^\lambda) + O\left(\frac{\lambda}{\nu}\right) &= \log_b(2^d) + O\left(\frac{s}{w}\right) + O\left(\frac{d}{\nu}\right) + O\left(\frac{s}{w \cdot \nu}\right) \\
&= \log_b(2^d) + O\left(\frac{d}{\nu}\right) + O\left(\frac{s}{w}\right) .
\end{aligned}$$

Now it suffices to observe that

$$\frac{s}{w} = \frac{s}{\lfloor p/2^{2^\nu+\nu} \rfloor} \le \frac{2s}{p/2^{2^\nu+\nu}} = 2 \cdot \frac{s}{p} \cdot 2^{2^\nu+\nu}.$$

(b) We use (a) with $\nu := \lfloor \log\log(pd/s) - 2 \rfloor \ge 1$. Then $d/\nu = O\left(d/(\log\log(pd/s))\right)$, and

$$2^{2^\nu+\nu} \cdot \frac{s}{p} \le 2^{2^{\nu+1}} \cdot \frac{s}{p} \le 2^{\frac{1}{2}\log(pd/s)} \cdot \frac{s}{p} = O\left(\frac{d}{\sqrt{pd/s}}\right) = O\left(\frac{d}{\log\log(pd/s)}\right) .$$

(c) Immediate from (b). $\square$

For the special case of formulas, i. e., circuits in which all gates have fan-out 1, a better additive error term can be achieved, as noted in the following.

THEOREM **5.6** *Let $n = 2^d$. If the Boolean function $f : \{0,1\}^n \to \{0,1\}$ can be represented by a formula F of depth d, then f can be evaluated by a CREW PRAM with p processors and $p+n$ memory cells in*

$$\varphi(n) + \left\lceil \frac{n}{p} \right\rceil + O\left(\frac{\log n}{\log\log n}\right)$$

steps.

Proof. We may assume w. l. o. g. that $p \leq n$ and that p is a power of 2. Assume first that $p = n$. In a first phase, the $2^{d-\lceil\sqrt{d}\rceil}$ subformulas of F are evaluated that correspond to the $2^{d-\lceil\sqrt{d}\rceil}$ subtrees of depth $\lceil\sqrt{d}\rceil$ at the bottom of F. Clearly, each such subformula can be evaluated in $\lceil\sqrt{d}\rceil + 1$ steps by $2^{\lceil\sqrt{d}\rceil}$ processors in the same number of memory cells of wordsize 1, even in an EREW fashion. Overall, n processors and cells are sufficient. We are left with the problem of evaluating a formula of depth $d' = d - \lceil\sqrt{d}\rceil$. Trivially, this formula is a circuit that can be arranged in levels of width at most $w := 2^{d'}$. By Lemma 5.4, for arbitrary ν this circuit can be evaluated by $p = w \cdot 2^{2^\nu+\nu}$ processors and $n + w \cdot 2^{2^\nu+\nu}$ memory cells in $\log_b(2^{d'}) + O(d'/\nu)$ steps. We choose $\nu = \lfloor\frac{1}{3}\log\log n\rfloor = \lfloor\frac{1}{3}\log d\rfloor$, and obtain that $w \cdot 2^{2^\nu+\nu} \leq 2^{d-\lceil\sqrt{d}\rceil} \cdot 2^{d^{1/3}+(1/3)\log d} \leq 2^d = n$ (for d sufficiently large) and, of course, $\log_b(d') \leq \varphi(2^d)$ and $O(d'/\nu) = O(d/\log d)$. In case $p < n$, we first reduce the size of the formula to p leaves by having each of the p subformulas of F of depth $d - \log p$ evaluated by one processor in n/p steps. Afterwards we proceed as before. $\square$

6. Many-valued formulas and circuits

Some of the previous results can be extended to k-valued formulas and circuits, i. e., devices that compute functions $\{0,\ldots,k-1\}^n \to \{0,\ldots,k-1\}^m$ for some $k \geq 3$ and are built from gates that compute functions

$$\otimes : \{0,\ldots,k-1\}^2 \to \{0,\ldots,k-1\}.$$

For CREW PRAMs that compute such functions the n arguments are stored in input memory cells of wordsize $\|k-1\|$, each argument in a separate cell. A component of the result is either given as a single value in an output cell of wordsize $\|k-1\|$ or, coded in binary, in $\|k-1\|$ cells of wordsize 1. We start with considering *simple* k-valued formulas, that is, functions $F(x_1,\ldots,x_n) = x_1 \otimes x_2 \otimes \cdots \otimes x_n$ for an associative operator $\otimes$ over $\{0,\ldots,k-1\}$, with values in $\{0,\ldots,k-1\}$. Such functions are direct generalizations of the functions AND_n, OR_n, and PARITY_n from the case $k = 2$. Theorem 4.1 may be generalized as follows.

THEOREM **6.1** *A simple k-valued formula F of size n can be evaluated by a CREW PRAM M with $p = n \cdot k^{(t+1)t/2}$ processors and $n \cdot (k^{t-1}+1)$ memory cells in $t = \varphi(n)+1$ steps, where each of the common memory cells of M, except the cells used for input and output, are of wordsize 1.*

The proof of this theorem is almost identical to the one for Theorem 4.1. The only difference is that instead of binary strings we use strings over the alphabet $\{0, \dots, k-1\}$ as names for cells and processors. If during the ith WRITE a group of processors wants to send a number $s \in \{0, \dots, k-1\}$ to a group of cells $\mathcal{C}$, the symbol $*$ is written to all cells in $\mathcal{C}$ with a name whose ith position differs from s, thus changing the state of these cells to "dead".

REMARK **6.2** We note that the result of such a computation can be written in binary in $\|k-1\|$ memory cells in one extra step. Indeed, we can prepare k different cells, all storing the same symbol. The processor that knows $F(\vec{x})$ after the READ of step $\varphi(n)+1$, marks one of these cells, namely the jth cell, where $j = F(\vec{x})+1$. During step $\varphi(n)+2$, another $k \cdot \|k-1\|$ processors read these cells in parallel ($\|k-1\|$ processors for each cell); during the following WRITE the $\|k-1\|$ processors that encountered the marked cell in parallel write the binary code of $F(\vec{x})$. In this way the algorithm uses only memory cells of wordsize 1 (except for the input cells).

REMARK **6.3** It is interesting to note that in many cases a Boolean formula that is not nicely balanced like the formulas of Theorem 5.6 can be considered as the restriction of a simple formula over a k-valued domain. (The well known fact that any unbalanced Boolean formula of size n, say, can be restructured to get an equivalent one of depth $O(\log n)$ does not help in constructing a fast CREW algorithm for the formula, because the constant factor in front of the logarithm in the depth bound (about a factor of 3) outweighs the saving from $\log n$ down to $\varphi(n)$.) As an example, consider the Horner type formula

$$F(x_1, \dots, x_n) = (\cdots((((x_1 \wedge x_2) \vee x_3) \wedge x_4) \vee x_5) \cdots \wedge x_n)$$

of size $n = 2m$ and depth $n-1$, which is extremely unbalanced. We define an associative operator $\otimes$ on the 3-valued domain $\{k, p, g\}$ by $u \otimes k = k$, $u \otimes g = g$, $u \otimes p = u$, for $u \in \{k, p, g\}$. Furthermore we define functions $G : \{0,1\}^2 \to \{k, p, g\}$ and $H : \{0,1\} \times \{k, p, g\} \to \{0,1\}$ by $G(x, 1) = g$, $G(1, 0) = p$, $G(0, 0) = k$ and $H(x, k) = 0$, $H(x, p) = x$, $H(x, g) = 1$, for $x \in \{0,1\}$. Then it is easy to see that

$$F(x_1, \dots, x_n) = H(x_1, G(x_2, x_3) \otimes G(x_4, x_5) \otimes \cdots \otimes G(x_{n-2}, x_{n-1}) \otimes G(x_n, 0)).$$

Therefore, by Theorem 6.1, one can evaluate $F(x_1, \dots, x_n)$ in $\varphi(n)+4$ steps on a CREW PRAM with subexponentially many processors and cells. The degree of F is n, hence at least $\varphi(n)$ steps are necessary to compute F, by Fact 1.4.

We generalize Theorem 5.1 to k-valued domains.

THEOREM **6.4** *Let $F(x_1, \dots, x_n) = x_1 \otimes \cdots \otimes x_n$ be a simple k-valued formula. Then there is an n-processor CREW PRAM M with n memory cells of wordsize $\|k-1\|$ that evaluates F in $\varphi(n) + O(\sqrt{\log n \cdot \log k}\,)$ steps.*

Proof. We may assume that $k \leq n$, since $\log n + 1$ steps are certainly sufficient, which is $O(\sqrt{\log n \cdot \log k}\,)$ for $k > n$. We proceed essentially as in the proof of Theorem 5.1. The input for stage i consists of $n_i = 2^{d-w_i}$ values $y_1^i, \dots, y_{n_i}^i$ in $\{0, \dots, k-1\}$ so that $F(x_1, \dots, x_n) = y_1^i \otimes \cdots \otimes y_{n_i}^i$. In stage i, the y_j^i, $1 \leq j \leq n_i$, are subdivided into $n_{i+1} := n_i/s_i$ blocks, or groups, of $s_i = 2^{u_i}$ consecutive values; the operation $\otimes$ is applied within each group to yield one new value y_j^{i+1}. This computation takes time $\varphi(s_i) + 1$, according to Theorem 6.1. The maximal group size we can afford without exceeding the processor bound can be easily calculated: Let u_i be the largest u such that $\lceil \log k \rceil \cdot \Phi(2^u) \le w_i$. Then for each group $s_i \cdot 2^{\lceil \log k \rceil \cdot \Phi(2^{u_i})} \ge s_i \cdot k^{\Phi(s_i)}$ processors are available, which is sufficient for applying Theorem 6.1. For this to work, we need $u_i \ge 1$, or $\lceil \log k \rceil \cdot \Phi(2^1) \le w_i$. In order to reach such a situation, we start with a preprocessing phase of $3\lceil \log k \rceil = O(\sqrt{\log n \log k}\,)$ steps in each of which simply adjacent pairs of values y_{2l-1}^i and y_{2l}^i are combined to form y_l^{i+1}. Thus, the first stage starts with $n_1 = 2^{d-w_1}$ values $y_1^1, \ldots y_{n_1}^1$ with $3\lceil \log k \rceil \le w_1$. Stage i is the last stage if $n_i \le 2^{u_i}$; in this case there is only one group of size $s_i = n_i$. The number of stages is denoted by m. The analysis now is very similar to the proof of Theorem 5.1. Let T be the computation time (without preprocessing). Then $T \le \varphi(n_1) + 2.34m \le \varphi(n) + 2.34m$. For estimating m, we prove just as in Theorem 5.1 that

$$u_i = \max\Big\{\, u \;\Big|\; \lceil \log k \rceil \cdot \Phi(2^u) \le \sum_{j=1}^{i-1} u_j + 3\lceil \log k \rceil \,\Big\}\ , \text{ for } 1 \le i \le m\ , \quad \text{and}$$

$$m = \min\Big\{\, i \mid 3\lceil \log k \rceil + \sum_{j=1}^{i} u_j \ge d \,\Big\}\,.$$

Lemma 5.2, applied with parameters $A = 2$, $q = 0$, $\eta = 0$, and $c = \lceil \log k \rceil$ yields $m = O(\sqrt{d \log k}\,) = O(\sqrt{\log n \log k}\,)$, as desired. $\square$

The last algorithm can be modified so that it uses only memory cells of wordsize 1. The simple idea is to store the intermediate results $y_1^i, \ldots, y_{n_i}^i$ passed from stage $i-1$ to stage i in binary encoding. For (sequentially) reading and writing these codes in each stage $2\|k-1\| = O(\log k)$ extra steps are sufficient. In this way, the computation time increases to $\varphi(n) + O\left(\sqrt{\log n \cdot \log^3 k}\right)$.

Finally, we generalize Theorems 5.5 and 5.6 to k-valued domains. The notion of a circuit consisting of gates with fan-in 2 that compute functions $\{0, \ldots, k-1\}^2 \to \{0, \ldots, k-1\}$ is an easy generalization of the binary case. (The prime example here is an arithmetic circuit over a ring or a field with k elements.) The definitions of the depth of such a circuit and of arranging the circuit in λ levels of width up to w (cf. Section 5) carry over directly, as does Lemma 5.3.

THEOREM **6.5** *Let C be a circuit of depth d consisting of s gates over the domain $\{0, \ldots, k-1\}$.*

(a) *Let $\nu \ge 1$ and p be arbitrary. Then C can be evaluated by a CREW PRAM with p processors and $s + p$ memory cells of wordsize $\|k-1\|$ in*

$$\varphi(2^d) + O\left(2^{\nu+1} \cdot k^{2^\nu} \cdot \frac{s}{p}\right) + O\left(\frac{d}{\nu}\right)$$

steps.

(b) *If p satisfies $\log\log(pd/s) - \log\log k \ge 3$, then there is a CREW PRAM with p processors and memory cells of wordsize $\|k-1\|$ that evaluates C in*

$$\varphi(2^d) + O\left(\frac{d}{\log\log(pd/s) - \log\log k}\right)$$

steps. (Note that this is $\varphi(2^d) + o(d)$ whenever k is fixed and $p = \omega(s/d)$.)

Proof. We may assume that $2^{\nu+1} \cdot k^{2^\nu} \leq p/2$, since otherwise even one processor can evaluate C in $2s = O\left(2^{\nu+1} \cdot k^{2^\nu} \cdot s/p\right)$ steps. Let $w := \lfloor p/(2^{\nu+1} \cdot k^{2^\nu}) \rfloor \geq 2$, and arrange C in $\lambda = d + \lfloor s/w \rfloor$ levels of width w (cf. Lemma 5.3). As in the proof of Lemma 5.4, the s gates are evaluated in stages $t = 1, 2, \ldots, \lambda/\nu$. Evaluating gate g at level l, $(t-1)\nu + 1 \leq l \leq t\nu$, with the values of all the gates in levels $L_0, L_1, \ldots, L_{l-(t-1)\nu}$ already available, amounts to computing the value of a function with $2^{l-(t-1)\nu}$ inputs from $\{0, \ldots, k-1\}$ and an output in $\{0, \ldots, k-1\}$. This can be done in $\varphi(2^{l-(t-1)\nu}) + 1 \leq \varphi(2^\nu) + 1$ steps by $k^{2^{l-(t-1)\nu}} \cdot 2^{l-(t-1)\nu}$ processors. (The method is practically the same as in the binary case: For each of the at most $k^{2^{l-(t-1)\nu}}$ possible inputs a team of $2^{l-(t-1)\nu}$ processors checks whether the actual input equals the input associated with the team. This is done by computing the OR of $2^{l-(t-1)\nu}$ bits. The unique successful team writes the result into the output cell.) The total number of processors used is bounded by $w \cdot 2 \cdot 2^\nu \cdot k^{2^\nu} = w \cdot 2^{\nu+1} \cdot k^{2^\nu} \leq p$. Proceeding exactly as in the proofs of Lemma 5.4 and Theorem 5.5 we may estimate the running time by $\varphi(2^d) + O\left(2^{\nu+1} \cdot k^{2^\nu} \cdot s/p\right) + O(d/\nu)$.

(b) Define $\nu := \lfloor \log\log(pd/s) - \log\log k - 2 \rfloor \geq 1$. Clearly, d/ν is bounded as required. Moreover,

$$\begin{aligned} 2^{\nu+1} \cdot k^{2^\nu} \cdot \frac{s}{p} &< k^{2^{\nu+1}} \cdot \frac{s}{p} \leq k^{2^{(\log\log(pd/s)-1)-\log\log k}} \cdot \frac{s}{p} \\ &= k^{(1/2)\cdot\log(pd/s)/\log k} \cdot \frac{s}{p} = \frac{d}{\sqrt{pd/s}} \\ &= O\left(\frac{d}{\log\log(pd/s) - \log\log k}\right). \end{aligned}$$

Thus, both O-terms in (a) can be bounded as claimed. □

In the case of k-valued formulas of depth d and size $n = 2^d$ (e. g., balanced arithmetic expressions over finite fields) the last result may be somewhat sharpened, in analogy to Theorem 5.6.

THEOREM **6.6** *Let $n = 2^d$ and let $k \leq 2^{\sqrt{d}/4}$. If the function $f : \{0, \ldots, k-1\}^n \to \{0, \ldots, k-1\}$ can be represented by a formula F of depth d (with arbitrary binary operators over $\{0, \ldots, k-1\}$), then f can be evaluated by a CREW PRAM with $p \leq n$ processors and $p + n$ memory cells of wordsize $\|k-1\|$ in*

$$\varphi(n) + \left\lceil \frac{n}{p} \right\rceil + O\left(\frac{\log n}{\log\log n - 2\log\log k}\right)$$

steps. (If $\log k = o(\sqrt{\log n})$ and $p = \omega(n/\log n)$, the number of steps is bounded by $\varphi(n) + o(\log n)$.)

Proof. We follow the proof of Theorem 5.6 and consider here only the case $p = n$. Subformulas of depth $\lceil\sqrt{d}\rceil$ may be evaluated in $\lceil\sqrt{d}\rceil + 1$ steps in the k-valued case as well. The remaining formula F' can be arranged in $d' = d - \lceil\sqrt{d}\rceil$ levels of width at most $w := 2^{d'}$. Let $\nu := \lfloor \frac{1}{2}\log d - \log\log k - 1 \rfloor$. Then $\nu \geq 1$ by the assumption $k \leq 2^{\sqrt{d}/4}$. By the proof of Theorem 6.5, $w \cdot 2^{\nu+1} \cdot k^{2^\nu}$ processors can evaluate F' in $\varphi(2^{d'}) + O(d'/\nu) = \varphi(n) + O(d/(\log\log n - 2\log\log k))$ steps. It remains to note that $w \cdot 2^{\nu+1} \cdot k^{2^\nu} \leq 2^{d-\lceil\sqrt{d}\rceil} \cdot k^{2^{\nu+1}} \leq 2^{d-\lceil\sqrt{d}\rceil} \cdot k^{\sqrt{d}/\log k} = 2^d = n$. □

The algorithms described in Theorems 6.4, 6.5, and 6.6 are unsatisfying in that they lead to computation times of $\varphi(n) + o(\log n)$ only if k is sufficiently small relative to n. Designing feasible algorithms that run in almost optimal time for arbitrary k seems to be difficult. However, in Section 8 we present a feasible algorithm for computing an important $(n+1)$-valued function, namely the sum of n bits, in almost optimal time.

7. Parallel prefix and addition

Let $\otimes$ be an associative operator over some domain. We say that a PRAM computes the parallel prefix for the product $x_1 \otimes x_2 \otimes \cdots \otimes x_n$ if on input $x_1, x_2, \ldots, x_n$ the PRAM computation results in the values of the n products

$$x_1, \quad x_1 \otimes x_2, \quad x_1 \otimes x_2 \otimes x_3, \quad \ldots, \quad x_1 \otimes \cdots \otimes x_n$$

stored in n fixed memory cells. Parallel prefix computation is a fundamental problem, which has been studied extensively for different computational models. Optimal realizations for the circuit model can be found in [19] and [13]. For unbounded fanin circuits, which relate to the CRCW PRAM model, see for example [10].

In this section we show that parallel prefix for k-valued domains can be computed on a CREW PRAM within practically the same complexity bounds as the single product $x_1 \otimes x_2 \otimes \cdots \otimes x_n$ (Theorem 6.4).

REMARK **7.1** In the construction of an adder for two n-bit numbers by Ladner and Fischer [22] a k-valued circuit for the parallel prefix problem is described that has depth $\log n$ and size $4n$. We could apply the general Theorem 6.5 to obtain a CREW PRAM algorithm for parallel prefix with n processors that runs in time $\varphi(n) + o(n)$ if $\log\log\log n - \log\log k = \omega(1)$, that means, $k = o(\log\log n)$. The direct construction presented in the following works for k with $\log k = o(\log n)$.

We start with a simple lemma.

LEMMA **7.2** *Let $\otimes$ be an associative operator $\{0,\ldots,k-1\}^2 \to \{0,\ldots,k-1\}$ over a k-valued domain. Then there is an EREW PRAM M with n processors and n memory cells of wordsize $\|k-1\|$ that computes the parallel prefix for $x_1 \otimes x_2 \otimes \cdots \otimes x_n$ in $\lceil \log n \rceil + 1$ steps.*

Proof. Without loss of generality we assume that $n = 2^d$. During the computation, the arguments $x_1, \ldots, x_n$ are divided into blocks; after step t these blocks consist of 2^{t-1} variables. Namely, for each $i \le n$, let $B_i^1 = \{x_i\}$, and for $t > 1$, $B_i^{t+1} = B_{2i-1}^t \cup B_{2i}^t$. So $B_i^t = \{x_{(i-1)\cdot 2^{t-1}+1}, x_{(i-1)\cdot 2^{t-1}+2}, \ldots, x_{i \cdot 2^{t-1}}\}$. Let $\prod B_i^t$ denote the product of the elements of B_i^t, that is, $x_{(i-1)\cdot 2^{t-1}+1} \otimes x_{(i-1)\cdot 2^{t-1}+2} \otimes \cdots \otimes x_{i\cdot 2^{t-1}}$. By prefix$(s, B_i^t)$ for $x_s \in B_i^t$ we denote $x_{(i-1)\cdot 2^{t-1}+1} \otimes x_{(i-1)\cdot 2^{t-1}+2} \otimes \cdots \otimes x_s$. The algorithm maintains the following invariant:

- After step t, for each $s \le n$, processor P_s knows $\prod B_i^t$ and prefix(s, B_i^t), where B_i^t is the block that contains x_s. Moreover, cell C_s stores $\prod B_i^t$.

The first step is simple: each processor P_s, for $s \le n$, reads from cell C_s. To describe step $t+1$ assume that the property above holds up to step t. Let $s \le n$ and $x_s \in B_i^{t+1} = B_{2i-1}^t \cup B_{2i}^t$. If $x_s \in B_{2i-1}^t$ then during step $t+1$ processor P_s reads from cell $C_{s+2^{t-1}}$, which stores $\prod B_{2i}^t$ (since $x_{s+2^{t-1}} \in B_{2i}^t$). If $x_s \in B_{2i}^t$, then P_s reads from cell $C_{s-2^{t-1}}$, which stores $\prod B_{2i-1}^t$. It is easy to check that this does not lead to a read conflict. Note that in both cases after the READ operation processor P_s knows $\prod B_{2i}^t$ as well as $\prod B_{2i-1}^t$. Then P_s computes $\prod B_i^{t+1}$ and writes it into cell C_s. Also, P_s computes prefix(s, B_i^{t+1}), knowing that prefix$(s, B_i^{t+1}) =$ prefix(s, B_{2i-1}^t), if $x_s \in B_{2i-1}^t$, and prefix$(s, B_i^{t+1}) = \prod B_{2i-1}^t \otimes$ prefix(s, B_{2i}^t) if $x_s \in B_{2i}^t$. The computation stops when $B_1^t = \{x_1, \ldots, x_n\}$, hence $t = \log n + 1$. During the WRITE phase of step t, each processor P_s writes prefix(s, B_1^t) into C_s, instead of $\prod B_1^t$. Thereby, after step t, we get the correct output. □

THEOREM **7.3** *Let $\otimes$ be an associative operator over a k-valued domain. The parallel prefix for $x_1 \otimes x_2 \otimes \cdots \otimes x_n$ can be computed by an n-processor CREW PRAM in $\varphi(n) + O(\sqrt{\log k \cdot \log n})$ steps using $2n$ memory cells of wordsize $\|k-1\|$.*

Proof. We may assume that $k \leq n$ and $n = 2^d$. (If $k > n$, use the algorithm of Lemma 7.2. If n is not a power of 2, take $n' = 2^{\lfloor \log n \rfloor}$ and compute the parallel prefix for $y_1 \otimes y_2 \otimes \cdots \otimes y_{n'}$, where $y_i = x_{2i-1} \otimes x_{2i}$ for $i \leq n - n'$ and $y_i = x_{i+(n-n')}$ for $n - n' < i \leq n'$. Then the products $x_1 \otimes \cdots \otimes x_j$, $1 \leq j \leq n$, can be obtained in two additional steps.) The idea of the algorithm is to divide the input variables into disjoint blocks and to compute the parallel prefix of each block. Then the input variables are partitioned into larger blocks and the parallel prefix of each block is computed, using the previous results. The second step is repeated until there is only one block consisting of all variables. At each time during the computation, cell C_r, for $r \leq n$, stores the value of the product $x_l \otimes x_{l+1} \otimes \cdots \otimes x_r$, where x_l is the beginning of the block that currently contains x_r. The remaining n memory cells are used for auxiliary data. Since after the last step there is a single block containing all variables, cell C_r stores the value of $x_1 \otimes x_2 \otimes \cdots \otimes x_r$, which is the correct result.

The computation consists of a preprocessing phase and of stages $i = 1, \ldots, m$. At the beginning of stage i, the following situation is given: The input variables are divided into blocks $B_{i,1}, B_{i,2}, \ldots, B_{i,n_i}$ of the same size, say 2^{w_i}; hence $2^{w_i} \cdot n_i = n$, or $n_i = 2^{d-w_i}$. For each block $B_{i,p} = \{x_{j_{i,p}}, x_{j_{i,p}+1}, \ldots, x_{j_{i,p+1}-1}\}$ there is a memory cell that stores

$$\prod B_{i,p} = x_{j_{i,p}} \otimes x_{j_{i,p}+1} \otimes \cdots \otimes x_{j_{i,p+1}-1}.$$

Let $y_p = \prod B_{i,p}$, for $p \leq n_i$. Further, cell C_r stores $x_{j_{i,p}} \otimes x_{j_{i,p}+1} \otimes \cdots \otimes x_r$, where $B_{i,p}$ is the block that contains x_r. Stage i consists of the following computation: M splits $y_1, y_2, \ldots, y_{n_i}$ into blocks $Y_p = \{y_{(p-1)\cdot s_i+1}, y_{(p-1)\cdot s_i+2}, \ldots, y_{p\cdot s_i}\}$ of some size s_i , where $1 \leq p \leq n_{i+1} := n_i/s_i$, and computes the parallel prefix of the values $y_1, y_2, \ldots, y_{n_i}$ within each block Y_p (details are given below). Each block $B_{i+1,p}$ consists of s_i blocks of the form $B_{i,j}$, namely,

$$B_{i+1,p} = B_{i,(p-1)\cdot s_i+1} \cup B_{i,(p-1)\cdot s_i+2} \cup \cdots \cup B_{i,p\cdot s_i}.$$

Of course, there are n_{i+1} such blocks $B_{i+1,p}$. Assume that x_r is in $B_{i,q} \subseteq B_{i+1,p}$. At the beginning of stage $i+1$, cell C_r must store the value of $x_{j_{i+1,p}} \otimes x_{j_{i+1,p}+1} \otimes \cdots \otimes x_r$. Note that

$$\begin{aligned} & x_{j_{i+1,p}} \otimes x_{j_{i+1,p}+1} \otimes \cdots \otimes x_r & (7.1) \\ = \; & \left(\prod B_{i,(p-1)\cdot s_i+1} \otimes \prod B_{i,(p-1)\cdot s_i+2} \otimes \cdots \otimes \prod B_{i,q-1}\right) \otimes \; (x_{j_{i,q}} \otimes x_{j_{i,q}+1} \otimes \cdots \otimes x_r) \\ = \; & (y_{(p-1)\cdot s_i+1} \otimes y_{(p-1)\cdot s_i+2} \otimes \cdots \otimes y_{q-1}) \; \otimes \; (x_{j_{i,q}} \otimes x_{j_{i,q}+1} \otimes \cdots \otimes x_r) \; . \end{aligned}$$

The value of $x_{j_{i,q}} \otimes x_{j_{i,q}+1} \otimes \cdots \otimes x_r$ is stored in cell C_r. The product $y_{(p-1)\cdot s_i+1} \otimes y_{(p-1)\cdot s_i+2} \otimes \cdots \otimes y_{q-1}$ is computed during stage i as one of the prefixes of block Y_p and stored in some fixed memory cell. Some processor reads this cell as well as cell C_r, computes the product $x_{j_{i+1,p}} \otimes x_{j_{i+1,p}+1} \otimes \cdots \otimes x_r$, according to (7.2), and stores the result in C_r.

It remains to fix the sizes of the blocks and the algorithm used within the stage, and to analyze the running time.

We start with a preprocessing phase, in which the parallel prefix is computed within blocks of length 2^{w_1}, where $w_1 = 1 + 3\lceil \log k \rceil$, by the algorithm of Lemma 7.2. This takes $2 + 3\lceil \log k \rceil = O(\sqrt{\log n \log k}\,)$ steps.

During stage i, $i \geq 1$, we must compute the parallel prefix within each block Y_p, for $1 \leq p \leq n_{i+1}$. Each single prefix product $y_{(p-1)\cdot s_i+1} \otimes y_{(p-1)\cdot s_i+2} \otimes \cdots \otimes y_{q-1}$ is computed by a separate group of processors and memory cells, by the algorithm of Theorem 6.1. For each such group, we employ $p(s_i) = s_i \cdot k^{\Phi(s_i)}$ processors (and fewer than $p(s_i)$ memory cells), where $\Phi(s_i) = (\varphi(s_i)+2)\cdot(\varphi(s_i)+1)/2$, as before. Since n_i products are to be computed, $n_i \cdot s_i \cdot k^{\Phi(s_i)}$ processors are required overall. If we let u_i be the largest integer u that satisfies

$$n_i \cdot 2^{u_i} \cdot 2^{\lceil \log k \rceil \cdot \Phi(2^{u_i})} \leq n\,,$$

that means, $u_i + \lceil \log k \rceil \cdot \Phi(2^{u_i}) \leq w_i$, then we may choose $s_i = 2^{u_i}$. By the preprocessing stage, we have $u_i \geq 1$ for all i. The last stage, m, is characterized as the minimal i with $n_i \leq 2^{u_i}$. Here, we choose $s_i = n_i$. The analysis of the running time (disregarding the preprocessing phase) is now practically the same as in Theorem 6.4. Stage i takes $(\varphi(s_i)+1)+2 = \varphi(s_i)+3 \leq \log_b(s_i)+4.34$ steps, and the overall time T can be estimated by $T \leq \varphi(n) + 4.34m$. In order to estimate m, we note that u_i is given by the recursion

$$u_i = \max\Big\{u \;\Big|\; u + \lceil \log k \rceil \cdot \Phi(2^u) \leq 1 + 3\lceil \log k \rceil + \sum_{j=1}^{i-1} u_j \Big\}\ , \text{ for } 1 \leq i \leq m\ , \quad \text{and}$$

$$m = \min\{i \mid 1 + 3\lceil \log k \rceil + \sum_{j=1}^{i} u_j \geq d\}\,.$$

In this situation, Lemma 5.2 is applicable (with $A = 2$, $\eta = 1$, $c = \lceil \log k \rceil$, and $q = 0$); it yields $m = O(\sqrt{c \cdot d}) = O(\sqrt{\log n \log k}\,)$. □

A CREW PRAM M adding two n-bit numbers gets the input bits in separate cells and has to generate the binary representation of the sum of these numbers, each bit in a separate cell.

Corollary **7.4** *For each n, there is an $(n+1)$-processor CREW PRAM with $2(n+1)$ memory cells of wordsize 2 that adds two n-bit numbers in $\varphi(n) + O(\sqrt{\log n}\,)$ steps.*

Proof. When computing in parallel a sum of two binary numbers, the main problem is to compute the carry bits. Once all carry bits are known, the sum can be computed in a few parallel steps. Let the carry propagation operator $\otimes : \{0,1,p\}^2 \to \{0,1,p\}$ be defined by

$$p \otimes x = x,\ 0 \otimes x = 0,\ 1 \otimes x = 1.$$

It is well known that this operator is associative and that the ith carry bit c_i equals $x_i \otimes x_{i-1} \otimes \cdots \otimes x_0$, where $x_0 = 0$ and for $i > 0$, $x_i = 1$, if both added numbers have 1's in position i;

$x_i = 0$, if there are two 0's in position i; and $x_i = p$ if there is a 0 and a 1 in position i. If we define $x \otimes' y = y \otimes x$, then $c_i = x_0 \otimes' x_1 \otimes' \cdots \otimes' x_i$. So, to compute the carry bits one has to compute the parallel prefix of $x_0 \otimes' x_1 \otimes' \cdots \otimes' x_n$. By Theorem 7.3, this can be done in $\varphi(n) + O(\sqrt{\log n}\,)$ steps by a CREW PRAM with $n+1$ processors and $2(n+1)$ memory cells of wordsize 2. In a few additional steps, this machine computes each bit of the sum. $\square$

REMARK **7.5** The currently best known construction of a circuit for adding two n-bit numbers was given by Krapchenko (see [34, p. 42]). This circuit has depth $\log n + O(\sqrt{\log n}\,)$ and size $O(n)$. Applying Theorem 5.5 to this circuit yields a CREW PRAM algorithm with running time $\varphi(n) + O(\log n / \log\log\log n)$. Comparing with Theorem 7.4 we see that the time bound is slightly better and that the structure of the algorithm is clearer for our direct construction.

8. Symmetric functions

A Boolean function is called symmetric if it only depends on the number of 1's in the input. In this section we describe feasible algorithms with an almost optimal running time for computing symmetric functions. An obvious way to compute such a function is to count the 1's in the input and then to determine the function value depending on the count. Thus, we start with studying the following task.

"Sum of n bits": For an input consisting of n bits $x_1, \ldots, x_n$ stored in n different memory cells, compute the binary representation of $\sum_{i=1}^{n} x_i$, and store it in $\|n\|$ memory cells (each bit in a separate cell).

We will describe a feasible algorithm for summing n bits, and apply it to the task of evaluating symmetric functions in time $\varphi(n) + o(\log n)$ with n processors.

REMARK **8.1** For the bit summation problem no circuits (with fan-in 2) of depth $\log n + o(\log n)$ are known. Thus, the CREW PRAM algorithm described here is faster by a constant factor than what what can be obtained by simulating the best known circuits for this problem. The same applies to the problem of computing arbitrary symmetric functions.

Let us first discuss two well-known methods, one for adding a sequence of bits, another one for adding a sequence of binary numbers. Although these methods are not good enough for our purpose, we will need them as subroutines; moreover, our main algorithm is a generalization of the second method described. The first method that we will describe is based on a well-known VLSI algorithm ([21], Section 1.1.4). It yields an algorithm for adding n bits that takes $O(\log n)$ steps on an EREW PRAM with n processors and n memory cells of wordsize 1.

LEMMA **8.2** *There is an n-processor EREW PRAM with n memory cells of wordsize* 1 *that computes the sum of $n = 2^d$ bits in $4d$ steps.*

Proof. We may assume that each READ phase consists of reading from two different cells. Such a machine, which we call a 2-READ PRAM, can be simulated by a usual EREW PRAM that makes twice as many steps. We prove the following by induction on k:

Claim: There is a 2-READ PRAM M_k with $2^k - 1$ processors and memory cells $C_0, \ldots, C_{2^k-1}$ of wordsize 1 that for an input $x_1, \ldots, x_{2^k}$ writes the binary representation $b_k b_{k-1} \cdots b_0$ of $\sum_{i=1}^{2^k} x_i$, starting at step k with the least significant bit b_0, and writing b_i into cell C_i at step $k+i$. The cell C_i is not used by M_k after step $k+i$. (The total number of steps is $2k$.)

M_1 reads the two input bits in step 1 and writes the two output bits in steps 1 and 2. So we assume that M_k exists and use it to construct M_{k+1}. We split the 2^{k+1} input bits into two groups of 2^k bits each and use two copies M'_k and M''_k of M_k, one for each group. Let $C'_0, \ldots, C'_{2^k-1}$, and $C''_0, \ldots, C''_{2^k-1}$ be the cells used by M'_k and M''_k, respectively. By C_i let us mean the cell C'_i, if $i < 2^k$, or the cell C''_{i-2^k}, if $i \geq 2^k$. In steps $t = k, k+1, \ldots, 2k$ the machines M'_k and M''_k produce the bits $b'_0, b'_1, \ldots, b'_k$ and $b''_0, b''_1, \ldots, b''_k$ of their (partial) sums. In addition, there is another processor P, whose task it is to add up the two sums produced by M'_k and M''_k, by using the standard "paper and pencil" method. Processor P keeps an internal variable c ("carry") initialized with 0; it is idle for the first k steps. In steps $t = k+1, \ldots, 2k+1$ processor P reads b'_i and b''_i, for $i = t - (k+1)$, the values written by M'_k and M''_k in step $t-1$ into cells C'_i and C''_i, and adds b'_i, b''_i and c, which results in a 2-bit number $s_1 s_2$. Then P writes s_2 as the next output bit into cell C_i (the same cell as C'_i) and puts $c := s_1$ (s_1 is the next carry bit). In step $t = 2k+2$ processor P writes c. Obviously, P outputs the bits of $\sum_{i=1}^{2^k} x_i$ in the proper order and with the required timing. Altogether $2(2^k - 1) + 1 = 2^{k+1} - 1$ processors are used. □

The second method that we describe is an adaptation of the "Wallace tree" construction for a circuit of depth $O(\log u + \log k)$ for adding u k-bit numbers. This construction is based on the idea of "carry-save addition" [33, 34].

LEMMA **8.3** (a) *If $b_i = b_{i,d-1} b_{i,d-2} \cdots b_{i,0}$, for $i = 1, 2, 3$, are three binary numbers with $b_1 + b_2 + b_3 < 2^d$, then there are two binary numbers $g_j = g_{j,d-1} g_{j,d-2} \cdots g_{j,0}$, for $j = 1, 2$, so that $b_1 + b_2 + b_3 = g_1 + g_2$ and so that there is an EREW PRAM of wordsize 1 with d processors that computes the bits $g_{j,e}$ from the bits $b_{i,f}$ in 4 steps.*
(b) *If u numbers b_i are given by their binary representations $b_{i,d-1} \cdots b_{i,0}$, for $0 \leq i < u$, and if $s = \sum_{i=0}^{u-1} b_i < 2^d$, then the binary representation $s_{d-1} \cdots s_0$ of s can be computed in $O(\log u + \log d)$ steps by an EREW PRAM with $d \cdot u$ processors and $d \cdot u$ cells of wordsize 1.*

Proof. (a) Let for $0 \leq l < d$ the sum $b_{1,2} + b_{2,l} + b_{3,l}$ have the binary representation $c_l s_l$. Define $g_{1,l} := s_l$ for $0 \leq l < d$, and $g_{2,l} := c_{l-1}$ for $1 \leq l < d$, and $g_{2,0} := 0$. (Note that $c_{d-1} = 0$.) Obviously, $g_1 + g_2 = b_1 + b_2 + b_3$. The method for calculating the numbers $g_{j,l}$ on an EREW PRAM is obvious.

(b) The computation proceeds in stages. At the beginning of each stage we have a collection of binary numbers with sum s. During a stage, we split these numbers into groups of three each, and then these three numbers are replaced by two new ones that have the same sum, as described in part (a). In that way, each stage reduces the number of the remaining numbers by a factor of $\frac{2}{3}$. Further, each stage needs at most $d \cdot u$ processors and $d \cdot u$ cells, and takes four steps. These stages are performed until only two numbers are left. It is not hard to see that the number of stages does not exceed $1 + \log_{3/2} u$. Finally, to add up the two d-bit numbers that remain after the last stage we use the algorithm of Corollary 7.4. The whole computation takes no more than $4 \cdot (1 + \log_{3/2} u) + O(\log d) = O(\log u + \log d)$ steps. □

Now we turn to the key result of this section.

THEOREM **8.4** *The sum of n bits can be computed by an n-processor CREW PRAM with n memory cells (of wordsize 1) in time $\varphi(n) + O(\log^{2/3} n \cdot \log\log n)$.*

Proof. Let $x_1, \ldots, x_n$ be the input bits, where w. l. o. g. $n = 2^d$. Our aim is to compute the binary representation of the sum $s = \sum_{i=1}^{n} x_i$ consisting of $\|n\|$ bits. The central idea of the algorithm is to generalize the addition method presented in Lemma 8.3 by using groups of more than 3 elements. Again, the algorithm proceeds in phases. The result of phase $i-1$ is a sequence $y_{i,1}, \ldots, y_{i,n_i}$ of binary numbers, each represented by $\|n\|$ bits, such that $\sum_{j=1}^{n_i} y_{i,j} = s$. The numbers $y_{i,j}$ are split into groups of a suitably chosen size a_i (as opposed to size 3 in Lemma 8.3), and from the a_i numbers in each group a set of $\|a_i\|$ new numbers is computed that has the same sum. By collecting the new numbers from all groups we obtain numbers $y_{i+1,1}, \ldots, y_{i+1,n_{i+1}}$, where n_{i+1} is substantially smaller than n_i. We have to show how to perform one such phase efficiently, i.e., in almost optimal time, and how to choose the parameters a_i and n_i in such a way that the numbers n_i decrease so fast that not too many phases are necessary.

In addition, the algorithm has a preprocessing phase and a postprocessing phase. The preprocessing phase is necessary to get n_1 numbers $y_{1,1}, \ldots, y_{1,n_1}$ with sum s where $\frac{n}{n_1}$, the number of processors per cell, exceeds a minimal value necessary to start the main procedure. The postprocessing phase begins after stage m, if n_m is sufficiently small. Then we add the remaining numbers using the procedure of Lemma 8.3(b). (We could also let the main procedure run until the end; however, this would unnecessarily complicate the analysis.) For the sake of simplicity assume throughout the proof that n is a sufficiently large number. (For small n, the algorithm from Lemma 8.2 is used.) We start by describing an efficient method for transforming a numbers given in binary into $\|a\|$ new numbers that have the same sum.

LEMMA **8.5** *Let $a \geq 3$ and $n = 2^d \geq 1$ be integers. Then there is a CREW PRAM with $\|n\| \cdot a \cdot a^{\Phi(a)}$ processors and memory cells of wordsize 1 that, when presented with the binary representations of numbers $v_0, \ldots, v_{a-1}$ with $\sum_{j=0}^{a-1} v_j \leq n$, in $\varphi(a) + 3$ steps computes the binary representations of numbers $z_0, \ldots, z_{\|a\|-1}$ that satisfy*

$$\sum_{l=0}^{\|a\|-1} z_l = \sum_{j=0}^{a-1} v_j \,. \tag{8.1}$$

Proof. Let $v_{j,d} \cdots v_{j,0}$ be the binary representation of v_i, for $j = 0, \ldots, a-1$. For each bit position $c \in \{0, \ldots, d\}$, consider the binary representation $b_{c,\|a\|-1} \cdots b_{c,0}$ of $b_c := \sum_{j=0}^{a-1} v_{j,c}$. We may rearrange these $\|a\| \cdot (d+1)$ bits so as to form $\|a\|$ numbers of length $\|n\| = d+1$ (see Figure 4 for an example). Namely, let z_l be the number with binary representation $b_{d-l,l} b_{d-l-1,l} \cdots b_{0,l} 0^l$. Then

$$\begin{aligned}
\sum_{j=0}^{a-1} v_j &= \sum_{j=0}^{a-1} \sum_{c=0}^{d} v_{j,c} \cdot 2^c = \sum_{c=0}^{d} \left(\sum_{j=0}^{a-1} v_{j,c} \right) \cdot 2^c = \sum_{c=0}^{d} b_c \cdot 2^c \\
&= \sum_{c=0}^{d} \sum_{l=0}^{\|a\|-1} b_{c,l} \cdot 2^{l+c} = \sum_{l=0}^{\|a\|-1} \left(\sum_{c=0}^{d} b_{c,l} \cdot 2^{l+c} \right) = \sum_{l=0}^{\|a\|-1} z_l.
\end{aligned}$$

(For the last equality recall that $b_c \cdot 2^c \leq \sum_{j=0}^{a-1} v_j \leq n$, and hence $b_{c,l} = 0$ for $l + c > d$, i. e., for $c > d - l$.) Thus, the numbers z_l satisfy eq. (8.1).

$$
\begin{array}{cccccccccl}
 & & v_{05} & v_{04} & v_{03} & v_{02} & v_{01} & v_{00} & = & v_0 \\
 & & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & & \vdots \\
 & & v_{45} & v_{44} & v_{43} & v_{42} & v_{41} & v_{40} & = & v_4 \\
\hline
 & & & & & b_{02} & b_{01} & b_{00} & = & \text{Sum of last column} \\
 & & & & b_{12} & b_{11} & b_{10} & & = & \text{Sum of next to last column} \\
 & & & b_{22} & b_{21} & b_{20} & & & = & \ldots \\
 & & b_{32} & b_{31} & b_{30} & & & & & \ldots \\
 & b_{42} & b_{41} & b_{40} & & & & & & \ldots \\
\multicolumn{2}{c}{\underbrace{b_{52} \quad b_{51}}_{=0}} & b_{50} & & & & & & = & \text{Sum of first column}
\end{array}
$$

$$\Downarrow \text{ Rearrange}$$

$$
\begin{array}{cccccccl}
b_{32} & b_{22} & b_{12} & b_{02} & 0 & 0 & = & z_2 \\
b_{41} & b_{31} & b_{21} & b_{11} & b_{01} & 0 & = & z_1 \\
b_{50} & b_{40} & b_{30} & b_{20} & b_{10} & b_{00} & = & z_0
\end{array}
$$

Figure 4: *Forming* $\|a\| = 3$ *numbers from* $a = 5$ *many by the procedure of Lemma 8.5, for* $d = 5$

By the definition, the binary representation of the numbers $z_0, \ldots, z_{\|a\|-1}$ can be obtained by rearranging the bits of the binary representations of $b_0, \ldots, b_d$. By Theorem 6.1 and Remark 6.2, each b_c can be computed in $\varphi(a) + 2$ steps by a CREW PRAM with $a \cdot a^{\Phi(a)}$ processors and cells of wordsize 1. So for $d + 1 = \|n\|$ numbers b_c we need $\|n\| \cdot a \cdot a^{\Phi(a)}$ processors (cells) altogether. In one additional step, the bits are rearranged so that we get binary representations of the numbers $z_0, \ldots, z_{\|a\|-1}$. □

We now describe and analyze the algorithm for summing n bits. It consists of a preprocessing phase, a main procedure (in m stages), and a postprocessing phase.

Preprocessing phase. The input bits are split into groups of 2^{w_1} elements with $w_1 := \lceil \log \|n\| \rceil + 4\Phi(16)$. The sum of the bits in each group is computed using the algorithm of Lemma 8.2. As a result, we get $n_1 := 2^{d-w_1}$ numbers $y_{1,1}, \ldots, y_{1,n_1}$ of binary length $\|n\|$ (filling up with leading zeroes) such that $\sum_{i=1}^{n_1} y_{1,i} = s$. For the preprocessing phase n processors, n cells, and $4w_1 = O(\log \log n)$ steps are sufficient.

Next we describe stage i of the main procedure, for $i \geq 1$. The number m of stages, as well as the parameters $a_i = 2^{u_i}$ and $n_i = 2^{d-w_i}$ (where $u_i, w_i \in I\!N$) will be defined by recursion in the course of the description.

Stage i**:**

Input: the numbers $y_{i,1}, \ldots, y_{i,n_i}$ with $\sum_{j=1}^{n_i} y_{i,j} = s$. Each $y_{i,j}$ is given by a binary representation with $\|n\|$ bits.

Computation: the numbers $y_{i,j}$ are split into n_i/a_i groups of a_i elements each. Using the method of Lemma 8.5, each group is replaced by $\|a_i\|$ new numbers with the same sum.

Output: Let $n_{i+1} = 2^{\lceil \log \|a_i\| \rceil} \cdot n_i/a_i$. Output the numbers $y_{i+1,1}, \ldots, y_{i+1,n_{i+1}}$ obtained by collecting the $\|a_i\|$ results from each group, padded with 0's to obtain n_{i+1} numbers.

LEMMA **8.6** (a) $n_{i+1} < 2 \cdot \|a_i\| \cdot n_i/a_i$;
(b) *stage i can be performed by $n_i \cdot \|n\| \cdot a_i^{\Phi(a_i)}$ processors and cells in $\varphi(a_i) + 3$ steps.*

Proof. (a) Obvious. (b) Each of the n_i/a_i groups uses $\|n\| \cdot a_i \cdot a_i^{\Phi(a_i)}$ processors (cells). Hence, $n_i \cdot \|n\| \cdot a_i^{\Phi(a_i)}$ processors (cells) are used altogether. □

Now we define the numbers n_i and a_i for $i > 1$ (hence also w_i, since $n_i = 2^{d-w_i}$). Assume that n_i has been determined already. By Lemma 8.6, $n_i \cdot \|n\| \cdot a_i^{\Phi(a_i)}$ processors (cells) are needed for performing stage i, thus we must have $n = 2^d \geq n_i \cdot \|n\| \cdot a_i^{\Phi(a_i)}$.

Define u_i as the largest number $u \in I\!N$ that satisfies $n \geq n_i \cdot \|n\| \cdot 2^{u \cdot \Phi(2^u)}$, and let $a_i = 2^{u_i}$. Note that $u_i \geq 4$ since $n/n_i = 2^{w_i} \geq 2^{w_1} \geq 2^{\log \|n\| + 4 \cdot \Phi(2^4)}$. The number m of stages of the main procedure is defined to be the minimal i such that $a_i \geq n_i$, i.e., $2^{u_i} \geq 2^{d-w_i}$, or $u_i + w_i \geq d$. In stage m, only one group of size n_m is formed; since $a_m \geq n_m$, the stage can be performed by the n processors. The result of the last stage are $\|n_{m+1}\| \leq \|n\|$ numbers that have sum s.

Postprocessing phase. The algorithm of Lemma 8.3 is applied to the up to $\|n\|$ numbers of $\|n\|$ bits with sum s that result from the main procedure. This algorithm outputs the binary representation of s after $O(\log \|n\|) = O(\log \log n)$ steps, using $\|n\| \cdot \|n\| = o(n)$ processors and cells.

It remains to analyze the main procedure. It is clear that n processors and memory cells are sufficient. The crucial step is to estimate m.

LEMMA **8.7** *The number m of stages of the main procedure is $O((\log n)^{2/3})$.*

Proof. From the construction, it is easily seen that the numbers u_i, w_i, for $i \geq 1$, obey the following recursive definition:

$$\begin{aligned} w_1 &= \lceil \log \|n\| \rceil + 4 \cdot \Phi(16)\,; \\ u_i &= \max\{u \mid \lceil \log \|n\| \rceil + u \cdot \Phi(2^u) \leq w_i\}\ , \text{ for } i \geq 1\,; \\ w_{i+1} &= w_i + u_i - \lceil \log \|2^{u_i}\| \rceil\ , \text{ for } i \geq 1\,. \end{aligned}$$

Since $w_i \geq w_1 = \lceil \log \|n\| \rceil + 4 \cdot \Phi(16)$, we have $u_i \geq 4$ for all i; hence $w_{i+1} \geq w_i + \frac{1}{4}u_i$ for all i, since $\lceil \log \|2^u\| \rceil \leq \frac{3}{4}u$ for $u \geq 4$. It is not clear how to analyze exactly the behaviour of the sequence u_i, $i \geq 1$. Instead, we use a lower bound that is easier to handle: We recursively define a sequence v_i, $i \geq 1$, by

$$v_i = \max\Big\{v \ \Big|\ 4v \cdot \Phi(2^{4v}) \leq 4\Phi(16) + \sum_{j=1}^{i-1} v_j\Big\}\ , \text{ for } i \geq 1\ .$$

A straightforward induction on i shows that $v_i \leq u_i$ and $w_1 + v_1 + \cdots v_{i-1} \leq w_i$ for all i. As m is the minimal i such that $w_i + u_i \geq d$, it follows that

$$m \leq \min\{i \mid w_1 + v_1 + \cdots v_i \geq d\}\ .$$

To the last expression we apply Lemma 5.2 with $q = 1$, $A = 16$, $\eta = 0$, and $c = 4$, which yields the bound $m = O((d^2 \cdot 4)^{1/3}) = O((\log n)^{2/3})$. □

We may now finish the time analysis. For $i \leq m$, stage i of the main procedure takes $\varphi(a_i) + 3$ steps Lemma 8.6(b). So, altogether stages 1 through m take $\sum_{i=1}^{m}(\varphi(a_i) + 3)$ steps. Both preprocessing phase and postprocessing phase take $O(\log \log n)$ steps. Thus, the total computation time T can be estimated as follows, using Lemmas 8.7 and 8.6(a):

$$\begin{aligned} T &\leq \sum_{i=1}^{m}(\varphi(a_i) + 3) + O(\log \log n) \\ &\leq \sum_{i=1}^{m} \log_b a_i + O(m) + O(\log \log n) \\ &\leq \sum_{i=1}^{m}(\log_b n_i - \log_b n_{i+1}) + \sum_{i=1}^{m} \log_b(2\|a_i\|) + O(\log^{2/3} n) \\ &\leq \log_b n_1 + m \cdot \log_b(2\|n\|) + O(\log^{2/3} n) \\ &\leq \varphi(n) + O(\log^{2/3} n \cdot \log \log n). \end{aligned}$$

This completes the proof of Theorem 8.4. □

Theorem 8.4 is the key for computing symmetric functions with n processors fast. We need another auxiliary result:

LEMMA **8.8** *An arbitrary Boolean function F of k arguments can be computed by a CREW PRAM with 2^k processors and 2^k memory cells of wordsize 1 in time $\varphi(k)+3$. Moreover, after the READ phase of step $\varphi(k)+3$, there is a processor that knows the whole input string.*

Recall that a CREW PRAM with $k \cdot 2^{k-1}$ processors can compute F in time $\varphi(k)+1$. On the other hand, with only k processors for almost all Boolean functions the computation takes $\Omega(k)$ steps provided that memory cells of wordsize 1 are used [4].

Proof. Let $k_1 = \lfloor k/2 \rfloor$ and $k_2 = n - k_1$. We split the input string $x_1, \ldots, x_k$ into substrings $\vec{v} = x_1, \ldots, x_{k_1}$ and $\vec{w} = x_{k_1+1}, \ldots, x_k$ of length k_1 and k_2, respectively. To $\vec{v}$ and $\vec{w}$ apply the algorithm of Fact 1.2, so that after the READ phase of step $\varphi(k_2)+1$ there is a processor $P_{\vec{v}}$ that knows $\vec{v}$ and a processor $P_{\vec{w}}$ that knows $\vec{w}$. Note that $k_1 \cdot 2^{k_1-1} + k_2 \cdot 2^{k_2-1} \leq k \cdot 2^{k_2} < 2^k$ processors (memory cells) have been used. We may assume that there are two sets $A_0, \ldots, A_{2^{k_1}-1}$ and $B_0, \ldots, B_{2^{k_2}-1}$ of memory cells, all initialized with 0. These cells may be used to code $\vec{v}$ and $\vec{w}$: processor $P_{\vec{v}}$ writes a "1" into cell A_s , where s is the number with binary representation $\vec{v}$, and processor $P_{\vec{w}}$ writes a "1" into cell B_r , where r is the number with binary representation $\vec{w}$. (This is done during the WRITE phase of step $\varphi(k_2)+1$.) Since we have $2^k = 2^{k_1} \cdot 2^{k_2}$ processors, we may assume that they are labeled $P_{i,j}$, for $0 \leq i < 2^{k_1}$, $0 \leq j < 2^{k_2}$. For each i, j, during the next two steps processor $P_{i,j}$ reads cells A_i and B_j. Only processor $P_{s,r}$ reads a "1" both times. Then $P_{s,r}$ knows that the input string is $\vec{v}\vec{w} = x_1, \ldots, x_k$ and may write the value $F(x_1, \ldots, x_k)$ into the output cell, during the WRITE phase of step $\varphi(k_2)+3 \leq \varphi(k)+3$. □

THEOREM **8.9** *An arbitrary symmetric function of n arguments can be computed by an n-processor CREW PRAM with n memory cells (of wordsize 1) in time*

$$\varphi(n) + O(\log^{2/3} n \cdot \log\log n) .$$

Proof. Let F be a symmetric function of n arguments. There is a function G of $k = \|n\|$ arguments such that if $y_1 \cdots y_k$ is the binary representation of $\sum_{i=1}^{n} x_i$, then $F(x_1, \ldots, x_n) = G(y_1, \ldots, y_k)$. The machine that computes F first finds the sum of the input bits using the algorithm of Theorem 8.4. Thus it obtains k bits $y_1, \ldots, y_k$ such that $F(x_1, \ldots, x_n) = G(y_1, \ldots, y_k)$. During the next two steps all processors read the cells storing y_1, y_2. Then the machine uses the algorithm of Lemma 8.8 for a function G_{y_1,y_2}, where $G_{i,j}(y_3, y_4, \ldots, y_k) = G(i, j, y_3, y_4, \ldots, y_k)$. The number of processors that we need for this phase is $2^{k-2} = 2^{\lceil \log(n+1) \rceil - 2} \leq 2^{\log(n+1)-1} = \frac{n+1}{2}$. Obviously, the total computation time is bounded by $\varphi(n) + O(\log^{2/3} n \cdot \log\log n) + O(\log\log n) = \varphi(n) + O(\log^{2/3} n \cdot \log\log n)$. □

9. Sorting algorithms

In this section we present algorithms for sorting n bits and sorting n binary numbers on small CREW PRAMs. In comparison to other sorting algorithms of logarithmic time complexity for networks and PRAMs [1, 8, 26], which sort arbitrary numbers, but have a running time bound $C \cdot \log n$ with a constant C much larger than 1, the method below achieves optimal time up to a lower order term for sorting bits. Finally, this method is extended to sorting arbitrary numbers given in binary representation.

THEOREM **9.1** *An n-processor CREW PRAM with n memory cells (of wordsize 1) can sort n bits in time $\varphi(n) + O(\log^{2/3} n \cdot \log\log n)$.*

Proof. The computation consists of three phases. During the first phase the sum of the input bits is computed and written in binary. By Theorem 8.4, this can be done in $\varphi(n) + O(\log^{2/3} n \cdot \log\log n)$ steps. During the second phase, the algorithm of Lemma 8.8 is applied in order to get a processor knowing the sum of the input bits, say a number $s \leq n$. This phase takes $\varphi(\|n\|) + 3 = O(\log\log n)$ steps. During the third phase, in $O(\log\log n)$ steps the number 1 is written into the cells $C_1, \ldots, C_s$, and the number 0 is written into the cells $C_{s+1}, \ldots, C_n$, thus getting the correct output. We describe the third phase in detail below. Given the claimed time bounds for the phases, it is clear that the whole computation takes at most $\varphi(n) + O(\log^{2/3} n \cdot \log\log n)$ steps.

The third phase consists of several stages. After each stage we get the correct output values written into all cells except for a small block of decreasing size. If at the beginning of a stage we start with an "unresolved" block B of size m, then during the stage the correct values are written into all cells of B except for some subblock of a size approximately equal to $\sqrt{m}$.

At the beginning of each stage we have the following situation:

- There is a block of adjacent memory cells B and a number $z \in \mathbb{N}$ such that in order to get the correct output the number 1 should be written into the first z cells of B and the number 0 into the rest of them.
- There are $|B|$ processors knowing that B is an "unresolved" block, one processor associated with each cell of the block. There is one processor that knows the number z.
- All memory cells except for those in B already contain the correct output values.

We will only describe the first stage. The other stages are essentially the same, except for the last one when the unresolved block has constant size and the output values are written sequentially by one processor. Without loss of generality we may assume that $n = 2^d$. At the beginning of the first stage, the "unresolved block" consists of all cells. Let $D_1 = 2^{\lceil d/2 \rceil}$, $D_2 = 2^{\lfloor d/2 \rfloor}$ and $s = (r-1) \cdot D_1 + s_1$, where $0 < s_1 \leq D_1$ (hence also $r \leq D_2$, since $s \leq n = D_1 D_2$). We divide the memory cells $C_1, \ldots, C_n$ into blocks of size D_1; namely, for $i \leq D_2$, block B_i consists of the cells $C_{(i-1)\cdot D_1+1}, C_{(i-1)\cdot D_1+2}, \ldots, C_{i \cdot D_1}$.

Let P be the processor that knows s. In order to get D_2 processors knowing r, the cells $C_1, \ldots, C_{D_2}$ are cleared so that they contain 0's. Then processor P writes a 1 into cell C_r. Next, cells $C_1, \ldots, C_{D_2}$ are read by the n processors, each cell exactly by D_1 of them. The processors that encounter a "1" know r. Then the blocks $B_1, \ldots, B_{r-1}$ are marked for filling with 1's and the blocks $B_{r+1}, \ldots, B_{D_2}$ for filling with 0's. Only block B_r might contain both 0's and 1's, so B_r cannot be filled at this stage of the computation. The marking of the blocks can be done in such a way that the symbol j is written into the first two cells of a block, if this block should be filled with symbol j. The first two cells of block B_r receive 0 and 1 to indicate that this block should not be filled now. The marking can be done in two steps, since we have $D_1 \geq D_2$ processors knowing r.

Then all n processors will be used again. For $i \leq n$, processor P_i reads the marked cells of the block containing cell C_i and according to the situation writes 0 or 1 into cell C_i, or does not write if C_i lies in the block marked with 0 and 1.

This is the end of the first stage. Note that B_r is the "unresolved" block, there are D_1 processors knowing that B_r is the "unresolved" block (namely, the processors knowing r) and that processor P knows s, hence also s_1. To obtain the correct output, a "1" is to be written into the first s_1 cells of B_r, and a "0" is to be written into the remaining cells of B_r. Hence we are in the correct situation for the beginning of the next stage.

Let $\gamma(m)$ be the time required by the procedure of the third phase for a block of m cells. By the construction, $\gamma(2^d) = \gamma(2^{\lceil \frac{d}{2} \rceil}) + 6$. It follows that $\gamma(2^d) = O(\log d)$, i. e., $\gamma(n) = O(\log \log n)$, as required. □

Next we show that binary numbers of arbitrary fixed length can be sorted in almost optimal time with small resources. As a preliminary step, we consider the problem of comparing two binary numbers.

FACT **9.2** *For each k, there is an EREW PRAM with k processors and $2k$ memory cells of wordsize 2 that compares two k-bit binary numbers in $\varphi(k) + 2$ steps.*

Essentially, the algorithm for comparing two numbers is the same as for computing the logical OR. We briefly sketch the idea. Let $a = a_{k-1} \cdots a_0$ and $b = b_{k-1} \cdots b_0$ be two binary numbers. Define operators $\otimes$ and $\odot$ as follows. For $x, y \in \{0, 1\}$,

$$x \otimes y \;=\; \begin{cases} g & \text{if } x > y, \\ e & \text{if } x = y, \\ s & \text{if } x < y. \end{cases}$$

For $z, z' \in \{g, e, s\}$,

$$z \odot z' \;=\; \begin{cases} z' & \text{if } z = e, \\ z & \text{otherwise.} \end{cases}$$

Clearly, comparing a and b can be done by computing the product $z_{k-1} \odot z_{k-2} \odot \cdots \odot z_0$, where $z_i = a_i \otimes b_i$, for $i = 0, \ldots, k-1$.

After computing the symbols z_i in two parallel steps, the product $z_{k-1} \odot \cdots \odot z_0$ is computed using the same method as that for the logical OR [9]. This is possible since the only properties of the OR used are the associativity of the operator $\vee$ and that

$$x \vee y \;=\; \begin{cases} y & \text{if } x = 0, \\ x & \text{otherwise.} \end{cases}$$

(The difference is that the operator $\odot$ is defined over a domain of *three* elements.) We leave the details to the reader.

THEOREM **9.3** *There is a CREW PRAM with $m^2 \cdot k$ processors and $m \cdot (m+1) \cdot k$ memory cells of wordsize 1 that sorts m binary numbers of length k in time*

$$\varphi(m \cdot k) + O(\log^{2/3} m \cdot \log \log m) \ .$$

Proof. Let k-bit numbers $a_1, \ldots, a_m$ be given. With each number a_j, for $1 \leq j \leq m$, we associate a group G_j of $m \cdot k$ processors and $m \cdot k$ cells $C_{j,1}, \ldots, C_{j,m\cdot k}$. The processors of G_j determine the position that is to be taken by a_j in the sorted string and copy a_j to this place. This works as follows:

(1) The processors of G_j copy the numbers $a_1, \ldots, a_m$ into the cells of G_j.

(2) For each $i \leq m$, the number a_j is compared with a_i in $\varphi(k) + 2$ steps by k processors. Let

$$b_{j,i} = \begin{cases} 1 & \text{if } a_j < a_i \text{ or } (a_j = a_i \text{ and } j \geq i), \\ 0 & \text{otherwise.} \end{cases}$$

For each $i \leq m$, the number $b_{j,i}$ is written into cell $C_{j,i}$. (Note that the number of 1's in the string $b_{j,1}, \ldots, b_{j,m}$ determines the position of a_j in the sorted string.)

(3) The bits $b_{j,1}, \ldots, b_{j,m}$ are sorted in $\varphi(m) + O(\log^{2/3} m \cdot \log\log m)$ steps; the resulting vector is written into cells $C_{j,1}, \ldots, C_{j,m}$.

(4) For each $i < m$, there are k processors that read the cells $C_{j,i}$ and $C_{j,i+1}$. In that way, some k processors detect the last cell $C_{j,s}$ containing a 1. These processors copy the binary representation of a_j into cells $C_{s,1}, \ldots, C_{s,k}$.

Clearly, the cells $C_{1,1}, \ldots, C_{1,k}; C_{2,1}, \ldots, C_{2,k}; \ldots; C_{m,1}, \ldots, C_{m,k}$ contain the correct output after phase (4). Phases (1) and (4) require a constant number of steps, so the whole computation takes $\varphi(k) + \varphi(m) + O(\log^{2/3} m \cdot \log\log m)$ steps. But $\varphi(k) + \varphi(m) = \log_b k + \log_b m + O(1) = \log_b(k \cdot m) + O(1) = \varphi(k \cdot m) + O(1)$. Hence the computation time is bounded as claimed. $\square$

Acknowledgment

We thank Marcin Kik for many discussions and for pointing out that the algorithm of Theorem 6.1 can be used as a subroutine by the algorithm of Lemma 8.5. The constructive comments of two anonymous referees are gratefully acknowledged. The results on the simulation of circuits were found following a question of one of the referees.

References

[1] M. Ajtai, J. Komlós, and E. Szemerédi, *Sorting in $c \log n$ parallel steps*, Combinat. **3** (1983), 1–19.

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

[3] P. Beame, M. Kik, and M. Kutyłowski, *Information broadcasting by Exclusive-Read PRAMs*, Parallel Processing Letters **4** (1994), 159–169.

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

[5] R. P. BRENT, *The parallel evaluation of general arithmetic expressions*, J. ACM **21** (1974), 201–208.

[6] J. BRUCK, *Harmonic analysis of polynomial threshold functions*, SIAM J. Discrete Math. **3** (1990), 168–177.

[7] S. BUBLITZ, U. SCHÜRFELD, B. VOIGT, AND I. WEGENER, *Properties of complexity measures for PRAMs and WRAMs*, Theoret. Comput. Sci. **48** (1986), 53–73.

[8] R. COLE, *Parallel Merge Sort*, SIAM J. Comput. **17** (1988), 770–785.

[9] 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.

[10] A. CHANDRA, S. FORTUNE, AND R. LIPTON, *Unbounded fan-in circuits and associative functions*, in Proc. 15th Annual ACM Symposium on Theory of Computing, 1983, 52–60.

[11] 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, 1990, 125–135.

[12] M. DIETZFELBINGER, M. KUTYŁOWSKI, AND R. REISCHUK, *Exact lower bounds for computing Boolean functions on CREW PRAMs*, J. Comput. Syst. Sci. **48** (1994), 231–254.

[13] F. FICH, *New bounds for parallel prefix circuits*, in Proc. 15th Annual ACM Symposium on Theory of Computing, 1983, 100–109.

[14] F. FICH AND A. WIGDERSON, *Towards understanding exclusive write*, in Proc. 1st ACM Symposium on Parallel Algorithms and Architectures, 1989, 76–82.

[15] R. M. KARP AND V. RAMACHANDRAN, Parallel algorithms for shared-memory machines, in J. van Leeuwen, ed., Handbook of Theoretical Computer Science, Vol. A, Algorithms and Complexity, pp. 869–941.

[16] M. KUTYŁOWSKI, *Fast algorithms for threshold functions on CREW PRAMs*, Technical Report, University of Wrocław, (Wrocław), Poland, January 1991.

[17] M. KUTYŁOWSKI, *Time complexity of Boolean functions on CREW PRAMs*, SIAM J. Comput. **20** (1991), 824–833.

[18] M. KUTYŁOWSKI AND R. REISCHUK, *Evaluating formulas on parallel machines without simultaneous writes*, Technical Report, Institut für Theoretische Informatik, TH Darmstadt, (Darmstadt), Germany, January 1990.

[19] R. E. LADNER AND M. J. FISCHER, *Parallel prefix computation*, J. ACM **27** (1980), 831–838.

[20] K. LANGE, *Unambiguity of Circuits*, Theoret. Comput. Sci. **107** (1993), 77–94.

[21] T. LEIGHTON, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann Publishers, San Mateo, California, 1991.

[22] R. LADNER AND M. FISCHER, *Parallel prefix computation*, J. ACM **27** (1980), 831–838.

[23] N. Nisan, *CREW PRAMs and decision trees*, in Proc. 21st Annual ACM Symposium on Theory of Computing, 1989, pp. 327–335.

[24] N. Nisan and M. Szegedy, *On the degree of boolean functions as real polynomials*, in Proc. 24th Annual ACM Symposium on Theory of Computing, 1992, pp. 462–467.

[25] 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.

[26] M. Paterson, *Improved sorting networks with $O(\log n)$ depth*, Algorithmica **5** (1990), 75–92.

[27] J. H. Reif, ed., *Synthesis of Parallel Algorithms*, Morgan Kaufmann Publishers, San Mateo, California, 1993.

[28] R. Smolensky, *Algebraic methods in the theory of lower bounds for Boolean circuit complexity*, in Proc. 19th Annual ACM Symposium on Theory of Computing, 1987, pp. 77–82.

[29] M. Szegedy, *Algebraic Methods in Lower bounds for Computational Models with Limited Communication*, Ph.D. Dissertation, University of Chicago, 1989.

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

[31] L. Stockmeyer and U. Vishkin, *Simulation of parallel random access machines by circuits*, SIAM J. Comput. **13** (1984), 402–422.

[32] U. Vishkin and A. Wigderson, *Trade-offs between depth and width in parallel computation*, SIAM J. Comput. **14** (1985), 303–314.

[33] C. S. Wallace, *A suggestion for a fast multiplier*, IEEE Transactions on Comput. **13** (1964), 14–17.

[34] I. Wegener, *The Complexity of Boolean Functions*, Wiley-Teubner, Stuttgart, 1987.