INFORMATION AND COMPUTATION 123, 72–110 (1995)

Fast Parallel Space Allocation, Estimation, and Integer Sorting*

HANNAH BAST AND TORBEN HAGERUP

Max-Planck-Institut für Informatik, D-66123 Saarbrücken, Germany

The following problems are shown to be solvable in $O(\log^* n)$ time with optimal speedup with high probability on a randomized CRCW PRAM using $O(n)$ space:

- Space allocation: Given n nonnegative integers $x_1, ..., x_n$, allocate n nonoverlapping blocks of consecutive memory cells of sizes $x_1, ..., x_n$ from a base segment of $O(\sum_{j=1}^{n} x_j)$ consecutive memory cells.
- Estimation: Given n integers in the range $1..n$, compute "good" estimates of the number of occurrences of each value in the range $1..n$.
- Semisorting: Given n integers $x_1, ..., x_n$ in the range $1..n$, store the integers $1, ..., n$ in an array of $O(n)$ cells such that for all $i \in \{1, ..., n\}$, all elements of $\{j: 1 \leqslant j \leqslant n \text{ and } x_j = i\}$ occur together, separated only by empty cells.
- Integer chain-sorting: Given n integers $x_1, ..., x_n$ in the range $1..n$, construct a linked list containing the integers $1, ..., n$ such that for all i, $j \in \{1, ..., n\}$, if i precedes j in the list, then $x_i \leqslant x_j$.

Moreover, given slightly superlinear processor and space bounds, these problems or variations of them can be solved in constant time with high probability. As a corollary of the integer chain-sorting result, it follows that n integers in the range $1..n$ can be sorted in $O(\log n/\log\log n)$ time with optimal speedup with high probability.

1. INTRODUCTION

This paper studies a number of problems that are of fundamental importance in parallel computing. Most of these have traditional, "exact" variants that are known not to possess fast parallel solutions. More precisely, computing the parity of n bits reduces to instances of these problems of size n, which, therefore, by the lower bound of Beame and Hastad (1989), cannot be solved faster than in $\Theta(\log n/\log\log n)$ time on a PRAM with any polynomial number of processors. The lower bound, stated only for deterministic algorithms, can be extended to randomized algorithms by means of a technique of Ajtai and Ben-Or (1984). Relaxing the problem definitions to allow approximate solutions, however, we are able to obtain very fast algorithms that run with optimal speedup on a randomized CRCW PRAM.

* Supported by the ESPRIT Basic Research Actions Program of the EC under Contracts 3075 and 7141 (Projects ALCOM and ALCOM II). A preliminary version of this paper was presented at the 23rd Annual ACM Symposium on Theory of Computing (STOC '91) under the title "Constant-Time Parallel Integer Sorting."

The first problem studied is that of *space allocation*, which we formalize as the *interval allocation* problem. Imagine that we are presented with n simultaneous requests, each of which is for a certain number of consecutive memory cells. Note that a request is not for specific memory cells, but merely indicates the number of cells needed. Abstractly speaking, such requests might originate with a collection of concurrently executing tasks, each of which needs a certain amount of working space for its computation. The present paper provides several concrete examples of situations where such requests arise naturally; many more can be found in the papers cited below. Given the set of requests, the goal is to satisfy each request, i.e., to supply the requesting agent with a private block of memory of the requested size.

We may view the allocated blocks as nonoverlapping subarrays of a single base array. The exact version of the interval allocation problem requires the size of the base array to exactly equal the sum of all requested sizes, and is clearly subject to the lower bound mentioned above. We must therefore relax this requirement, but still want to insist that not too much space be wasted. For reasons similar to those that motivate the use of the O-notation, we require the size of the base array to be at most a constant factor larger than the sum of the requested sizes. With this convention, we are able to solve interval allocation problems of size n in $O(\log^* n)$ time with optimal speedup with high probability. As shown by MacKenzie (1992), this is as fast as possible for any algorithm that uses no more than n processors. A variant of interval allocation called *interval marking* is a natural formalization of the (vaguely defined) *processor allocation* problem, which adds to the importance of the interval allocation problem. Part of the development of our interval allocation algorithm took place in a dialog with Joseph Gil, and a result similar to ours was reported in (Gil *et al.*, 1991); details are given in Section 7.

The second problem studied is that of *profiling*. We are here given an array of n keys, and the task is to determine the multiplicity (i.e., the number of occurrences) of each value represented among the keys. The exact version of the problem asks for the exact multiplicities and, again, is clearly subject to the lower bound of $\Omega(\log n/\log\log n)$. We therefore content ourselves with approximate counts. We

0890-5401/95 $12.00

assume that the values represented among the keys are integers in a range $1..m$, where m is at most linear in n. If this is not the case initially, static hashing can frequently be used to map the original values injectively to a sufficiently small range of integers, after which approximate counts can be computed using our algorithms and associated with the original values (see (Bast and Hagerup, 1991) and (Bast *et al.*, 1992) for the best known results on static hashing on the CRCW PRAM). The output hence takes the form of m nonnegative integers $\hat{b}_1, ..., \hat{b}_m$, where $\hat{b}_i$ is an estimate of the number b_i of occurrences of the value i, for $i = 1, ..., m$.

We study two different variants of the profiling problem. In the first of these, the number of different values is assumed to be much smaller than the number of keys; specifically, $m = O(n^{1-\delta})$, for some fixed $\delta > 0$. In these circumstances, constant time and n processors suffice, with high probability, to compute what we call a *fine-profile*, i.e., a sequence $\hat{b}_1, ..., \hat{b}_m$ with $b_i \leqslant \hat{b}_i \leqslant Kb_i$, for some constant $K \geqslant 1$ and for $i = 1, ..., m$. This simple result furnishes a basic tool used in many of our other algorithms.

The second variant of the profiling problem is concerned with the case $m = n$, which is of special interest and importance (see below). It requires the estimates $\hat{b}_1, ..., \hat{b}_n$ to be independent random variables (note that $\hat{b}_1, ..., \hat{b}_m$ are random variables because the execution of one of our (randomized) algorithms constitutes a random experiment; the input is considered to be fixed). Given the difficulty of analysis often caused by a lack of independence, this is a reasonable property for which to ask. As concerns the accuracy of the estimates, we require on the one hand that $\sum_{i=1}^{n} \hat{b}_i = O(n)$, a natural condition, and on the other hand that $\Pr(b_i > a\hat{b}_i) \leqslant 2^{-a}$, for $i = 1, ..., n$ and for all $a \geqslant 1$ (i.e., the probability of an estimate being a times too small decreases exponentially in a), a less natural condition that represents a compromise between what we would ideally like and what we can easily compute. We show that estimates $\hat{b}_1, ..., \hat{b}_n$ with this property, called a *coarse-profile*, can be computed in $O(\log^* n)$ time with optimal speedup with high probability.

The third problem studied is that of *semisorting* (the term was taken from (Valiant, 1990)). To semisort a sequence of objects, each with a distinguished key, is to rearrange the objects so that all objects with a common key occur together. We assume that the keys are integers in the range $1..n$; as above, static hashing can often be used to enforce this condition if it is not satisfied initially. The lower bound of $\Omega(\log n/\log\log n)$ applies to semisorting, as defined so far, so we relax the definition by allowing the output to be given in the form of a *padded sequence* of size $O(n)$; i.e., $O(n)$ special *null* objects are allowed to intervene in arbitrary positions between the n objects that form the actual semisorted sequence. Our result is that semisorting problems of size n can be solved in $O(\log^* n)$ time with optimal speedup with high probability. The proof is quite involved and makes crucial use of the results obtained for the second variant of profiling—the condition $\Pr(b_i > a\hat{b}_i) \leqslant 2^{-a}$ turns out to be exactly what is needed. We extend the semisorting result to *strong semisorting*, which requires that all occurrences of a key of multiplicity b appear in a subarray of the output array of size $O(b)$.

Semisorting has many and diverse uses. Our result on strong semisorting directly provides one of our best profiling results, namely a fine-profile for the case $m = n$. Another immediate application is to integer *chain-sorting*. General chain-sorting takes as input n keys drawn from a totally ordered universe and makes each key point to the next larger key, if any (with an arbitrary total order imposed by the algorithm on each set of keys of a common value); i.e., the keys are stored in sorted order in a linked list. We consider the chain-sorting problem with integer input keys drawn from the range $1..n$. In contrast with what is the case for profiling and semisorting, a preprocessing based on hashing, which is a nonmonotonic operation, does not enable our integer chain-sorting algorithm to cope with more general input keys; allowing only keys in the range $1..n$ therefore is a true restriction. In recognition of this fact, we continue to use the term "integer chain-sorting," rather than simply "chain-sorting." Note also that the lower bound of Beame and Hastad does not apply to chain-sorting, even with no restriction on key values. On the other hand, the well-known lower bound of $\Omega(n \log n)$ for (randomized) comparison-based sequential sorting, which holds also for chain-sorting, implies that our result, $O(\log^* n)$ time with optimal speedup with high probability, does not extend from integer chain-sorting to general chain-sorting, which allows only pairwise comparisons. The first $O(\log^* n)$-time integer chain-sorting algorithm was given by Gil *et al.* (1991); their argument, however, is very sketchy, and their result is slightly weaker than ours. As a rather trivial by-product of our fast integer chain-sorting algorithm, we are able to improve the best previous result on (standard) randomized sorting of n integers in the range $1..n$: We show that this problem can be solved in $O(\log n/\log\log n)$ time with optimal speedup with high probability. Slightly weaker results were found independently by Matias and Vishkin (1991) and Raman (1991); see Section 10.

More substantial applications of our semisorting result were reported elsewhere. Semisorting is used in (Hagerup, 1992a, 1992b) to simulate stronger PRAM variants on the weaker TOLERANT PRAM; semisorting there serves to bring together all write requests pertaining to a common memory cell. Hagerup and Katajainen (1993) employ semisorting in the construction of the Voronoi diagram of n random sites drawn independently from the uniform distribution over the unit square; a grid divides the unit square into approximately n cells, and the set of sites in each cell is computed by means of semisorting. Our result also allows a significant simplification of the hashing scheme of (Bast and

Hagerup, 1991). In (Hagerup and Raman, 1992), finally, semisorting is used for a variety of different purposes.

From a different point of view, the present paper explores the power flowing from a combination of three new techniques in algorithm design and analysis: First, the "log-star" technique introduced by Raman (1990) and developed further by Matias and Vishkin (1991). Second, randomized "scattering" procedures for estimating various quantities crudely, but rapidly. Third, the analysis of randomized algorithms using martingale theory, which is not new, but which in the past has not been used as often as it deserves. A less detailed and more accessible account of most of the material in this paper can be found in (Hagerup, 1992b); the reader may want to study that paper before taking on the present one.

The structure of the paper is as follows: After some preliminaries in Section 2, Section 3 introduces various concepts under the general heading of "scattering" and lists some of their basic properties. Section 4 deals with a special case of interval allocation called *compaction*, and Section 6 extends this to so-called *colored compaction*. Section 5 presents first results for the fine-profiling problem, and Section 7 uses the results of Sections 5 and 6 to solve the interval allocation problem. Sections 8 and 9 are devoted to coarse-profiling and semisorting, respectively, and Section 10 describes applications of semisorting. Section 11, finally, studies the consequences of allowing slightly superlinear processor and space bounds. Every section uses essentially all sections before it, so that it is difficult to read sections out of context.

2. PRELIMINARIES

A *CRCW PRAM* (concurrent-read concurrent-write parallel random access machine) is a synchronous parallel machine with processors numbered 1, 2, ... and with a global memory that supports concurrent (i.e., simultaneous) access to a single cell by arbitrary sets of processors. The semantics of concurrent writing can be defined in many ways. Accordingly, many different variants of the CRCW PRAM, each distinguished by a different rule for the resolution of write conflicts, have been introduced; see, e.g., (Chlebus *et al.*, 1989; Hagerup and Radzik, 1990; Hagerup, 1992a) for definitions of many of these models and for discussion of the relationships between them. The following two write conflict resolution rules and corresponding variants are relevant to the present paper:

ARBITRARY (Shiloach and Vishkin, 1982): If two or more processors attempt to write to a given cell in a given step, then one of them succeeds, but there is no rule assumed to govern the selection of the successful processor.

TOLERANT (Grolmusz and Ragde, 1987): If two or more processors attempt to write to a given cell in a given step, then the contents of that cell do not change.

It is easy to see that the ARBITRARY PRAM is (not necessarily strictly) stronger than the TOLERANT PRAM in the sense that one step of a TOLERANT PRAM can be simulated by a constant number of steps on an ARBITRARY PRAM with the same number of processors and memory cells. In fact, most CRCW PRAM models commonly considered are stronger than the TOLERANT PRAM in this sense. We employ the TOLERANT model throughout the paper, with the sole exception of Lemmas 3.4(b) and 3.5 and Theorem 11.6 and their proofs, which use the ARBITRARY model. The most direct implementation of some of our other algorithms, however, assumes the ARBITRARY model, and we have to put in an extra effort in order to derive a solution for the weaker TOLERANT PRAM. Since we expect the distinction between different variants of the CRCW PRAM to be of little concern to many readers, we try to make it possible to skip material that deals only with the translation between models. All of our results hold also for the PRIORITY PRAM, which is even stronger than the ARBITRARY PRAM, while they do not extend immediately to the COMMON PRAM, which cannot simulate the TOLERANT PRAM in a step-by-step fashion without loss (Grolmusz and Ragde, 1987).

Consider the following assertion: "Every problem that can be solved in τ time steps with p processors can also, for every given $k \in \mathbb{N}$, be solved in $O(k\tau)$ time with $\lceil p/k \rceil$ processors." A simple but important simulation shows the assertion to hold for the ARBITRARY PRAM: Each physical processor simulates up to k virtual processors in a step-by-step fashion. We express this by saying that the ARBITRARY PRAM is *self-simulating* and sometimes use the word "processor" to denote a virtual processor in the sense of this simulation. The number of *operations* executed by a parallel algorithm that uses τ time steps and p processors is defined to be its time-processor product $p\tau$. By the simulation above, we always have $p\tau = \Omega(T)$, where T is the sequential complexity of the problem solved by the algorithm. Accordingly, the parallel algorithm is said to have *optimal speedup* or to be *optimal* if $p\tau = O(T)$. Because of the self-simulating property, if a problem can be solved on an ARBITRARY PRAM using t time steps and q operations, then it can also, for every given $\tau \geq t$, be solved in $\Theta(\tau)$ time using $O(q + \tau)$ operations; i.e., the algorithm can be slowed down without loss. This makes it convenient to express the performance of the algorithm by giving the pair (t, q) of minimum computation time and number of operations. In contrast with all other commonly considered PRAM variants, the TOLERANT PRAM is not known to be self-simulating. Since it is still important to know the extent to which a particular algorithm can be slowed down (see below), we are forced to indicate this explicitly, typically in a statement of the form "For all $\tau \geq \log^* n$, $O(\tau)$ time and $\lceil n/\tau \rceil$ processors suffice to ...." We advise the reader to interpret such a statement as "The time is $O(\log^* n)$, and the algorithm is optimal and

can be slowed down." Note that if an algorithm consists of l parts with (minimal time, number of operations) performance pairs $(t_1, q_1), \ldots, (t_l, q_l)$ and if each part can be slowed down, in the sense above, then the whole algorithm has a performance pair (t, q), where $t = O(\sum_{i=1}^{l} t_i)$ and $q = O(\sum_{i=1}^{l} q_i)$. It can also be shown that when $k \in \mathbb{N}$ is a constant, we can always reduce the number of processors from p to $\lceil p/k \rceil$, even on the TOLERANT PRAM, without increasing the processing time by more than a constant factor and the space requirements by more than $O(p)$. We shall freely use this observation, which was also made in (Gil, 1990) in a special case.

The majority of our algorithms are randomized. Randomized algorithms are customarily divided into *Monte Carlo* algorithms, which may occasionally err, and *Las Vegas* algorithms, which never err, but which may either take a long time to produce a (correct) result, or finish on time without producing any result—it is easy to transform any Las Vegas algorithm from one of these modes of operation to the other. In all cases, the analysis of a randomized algorithm bounds the probability of the relevant undesirable behavior, which we call the *failure probability* (for a Las Vegas algorithm, this is a slight misnomer).

A Las Vegas algorithm is clearly more desirable than a Monte Carlo algorithm, since it is trivial to run a Las Vegas algorithm as a Monte Carlo algorithm: If the algorithm has not produced any result within a suitable response time, abort it (if it is still running) and output an arbitrary value. The distinction between Monte Carlo algorithms and Las Vegas algorithms will be crucial at one point of our exposition. At any rate, although this is not always done, we believe that it is important to classify each randomized algorithm as either Monte Carlo or Las Vegas. We will do so by appending the appropriate one of "(Monte Carlo)" and "(Las Vegas)" to the bound on the failure probability given for each algorithm.

Informally, a randomized algorithm can be formulated as a Las Vegas algorithm whenever its output can be verified using negligible resources, either after the fact or by the algorithm itself—this is because the algorithm can be executed repeatedly until some output passes the verification. Whenever we classify an algorithm as a Las Vegas algorithm, it will be easy to see that such verification is possible, and we will not demonstrate this explicitly.

As usual, $E_1 = O(E_2)$, where E_1 and E_2 are expressions, means that there is a constant $c > 0$ such that $E_1 \leq cE_2$. Note that we require this relation to hold for all legal values of the parameters occurring in E_1 and E_2, not just for sufficiently large values of these parameters. The constant c is independent of all other parameters, except that it may depend on quantities that are explicitly qualified as "fixed" (in the present paper, such quantities are always denoted by the symbols δ and μ). The meaning of $E_1 = \Omega(E_2)$ is defined analogously.

In order to make many proofs more readable, we make extensive use of the notion of a *negligible probability*. What constitutes a negligible probability depends on the context. Is the goal, e.g., to show that some event occurs with probability $2^{-n^{\Omega(1)}}$, then in the proof we can ignore any polynomial (in n) number of probabilities of the form $2^{-\Omega(n^{\varepsilon})}$, for arbitrary $\varepsilon > 0$, since for sufficiently large values of n the sum of such probabilities will be bounded by $2^{-n^{\varepsilon'}}$, for suitable $\varepsilon' > 0$; we here rely on the fact that all problems considered in the paper become trivial if the problem size n is bounded by a constant. An event that occurs *with high probability* is the complement of an event of negligible probability. We often tacitly assume that such events always occur. Whenever we speak of "choosing at random," we mean choosing from the uniform distribution and independently of other such choices. We assume processors to be equipped with unit-time operations for integer addition, subtraction, multiplication and division with remainder, for computing 2^s, for every given $s \in \mathbb{N}$, and for choosing a random integer from the set $\{1, \ldots, s\}$, for every given $s \in \mathbb{N}$. As a minimum, we assume that the available operations can be executed in constant time for integer operands and results of absolute value bounded by $n + m + p$, where n is the input size, m is the largest absolute value of an input number, and p is the number of processors of the machine under consideration; i.e., a (standard) logarithmic word length suffices.

When nothing else is stated, arrays are assumed to be one-dimensional. Given an array A, we denote by $|A|$ its *size*, i.e., the number of cells in A. Although, in principle, a memory cell contains a single integer, we often find it convenient to pretend that a cell can contain an entire record of an arbitrary, but constant, number of (integer) fields; achieving this is simply a matter of considering a constant number of cells as a unit, also called a cell. When we state that a problem can be solved by a certain number of processors or speak of allocating a certain number of processors to some task, we always assume the processors to be consecutively numbered. Without stating this explicitly, we also assume that each processor "comes equipped with" a cell indexed by its processor number in a suitable array shared by all processors, which can be used for coordination between processors working on a common task. One consequence of this is that our processor bounds are always dominated by our space bounds.

We use "log" to denote the binary logarithm function. For $k = 0, 1, \ldots, \log^{(k)}$ denotes k-fold application of the function $x \mapsto \max\{\log x, 1\}$, i.e., $\log^{(0)} x = x$ and $\log^{(k)} x = \max\{\log \log^{(k-1)} x, 1\}$, for all $x > 0$ and all $k \in \mathbb{N}$. For $n \in \mathbb{N}$, $\log^* n = \min\{k \in \mathbb{N} : \log^{(k)} n = 1\}$. Although extracting logarithms is not one of our standard operations, we will assume that the function $x \mapsto \lfloor \log x \rfloor$ can be evaluated in constant time by a single processor, for $x \in \{1, \ldots, n\}$. This is justified by an observation of Hagerup and Radzik (1990), who show that a table realizing this function on the domain

in question can be constructed in constant time with n processors. As a consequence, $\lfloor \log^{(k)} n \rfloor$ can be computed in $O(k)$ time by a single processor, for arbitrary given $k \in \mathbb{N}$. It is also easy to see that for any given rational number $q \leqslant 1$, the function $x \mapsto \lfloor x^q \rfloor$ can be evaluated in constant time with n processors, for $x \in \{1, ..., n\}$ (details are given in (Hagerup, 1992c)).

Some of the constants appearing in our proofs are very large. This is not evidence of a true problem, but merely reflects a decision never to add to the complexity of an argument in order to obtain smaller constants. In particular, we make frequent use of the very crude estimates $2^x \geqslant x$, for all $x \geqslant 0$, and $\lceil x \rceil \leqslant 2\lfloor x \rfloor$, for all $x \geqslant 1$. We expect that a less generous analysis would yield reasonable constant factors.

Our probabilistic analysis is based on the two lemmas below. Lemma 2.1 states various inequalities commonly known as Chernoff bounds.

LEMMA 2.1. *For every binomially distributed random variable S, the following holds:*

(a) *For all $z \geqslant 2E(S)$, $\Pr(S \geqslant z) \leqslant e^{-z/6}$;*

(b) $\Pr(S \leqslant E(S)/2) \leqslant e^{-E(S)/8}$;

(c) *For all $z > 0$, $\Pr(S \geqslant z) \leqslant (eE(S)/z)^z$.*

Proof. Part (a) with $z = 2E(S)$ as well as parts (b) and (c) are well-known and proved, e.g., in (Hagerup and Rüb, 1990). In order to show the general form of part (a), assume that S is the number of heads in m independent tosses of a coin with probability p of heads and, without loss of generality, that $z \leqslant m$. Let $Z_1, ..., Z_m$ be independent random variables with range $\{0, 1, 2\}$ and with $\Pr(Z_i = 1) = p$ and $\Pr(Z_i \in \{1, 2\}) = z/(2m)$ $(\geqslant p)$, for $i = 1, ..., m$. Then $S_1 = |\{i: 1 \leqslant i \leqslant m \text{ and } Z_i = 1\}|$ is distributed as S, $S_2 = |\{i: 1 \leqslant i \leqslant m \text{ and } Z_i \geqslant 1\}|$ is binomially distributed with $E(S_2) = z/2$, and $S_2 \geqslant S_1$. Then, however, by the special case of part (a) already established, $\Pr(S \geqslant z) = \Pr(S_1 \geqslant z) \leqslant \Pr(S_2 \geqslant z) = \Pr(S_2 \geqslant 2E(S_2)) \leqslant e^{-z/6}$. ∎

The following fact is implied by Azuma's inequality (see (Bollobás, 1987) or (McDiarmid, 1989)). Corollary 2.3 expresses the special form of Lemma 2.2(a) that we shall most often use.

LEMMA 2.2. *Let $m \in \mathbb{N}$, let $Z_1, ..., Z_m$ be independent random variables with finite ranges, and let S be an arbitrary real function of $Z_1, ..., Z_m$ with $E(S) \geqslant 0$. Assume that S changes by at most c in response to an arbitrary change in a single Z_i. Then*

(a) *For every $z \geqslant 2E(S)$, $\Pr(S \geqslant z) \leqslant e^{-z^2/(8c^2m)}$;*

(b) $\Pr(S \leqslant E(S)/2) \leqslant e^{-(E(S))^2/(8c^2m)}$.

COROLLARY 2.3. *Under the assumptions of Lemma 2.2,*

$$\Pr(S \geqslant \max\{2E(S), 4cm^{1/2+\varepsilon}\}) \leqslant 2^{-m^\varepsilon}$$

for all $\varepsilon > 0$.

In later applications, we write "by a Chernoff bound" instead of "by Lemma 2.1," and "by a martingale argument" rather than "by Corollary 2.3."

When we state that an algorithm makes at most m random choices, a change in one of which affects some real quantity S by at most c, what we mean is that S is determined somehow by an execution of the algorithm, and that the execution is deterministically given by the input, except that it may also be influenced by at most m *independent* random quantities computed by the algorithm, a change in one of which (with the other random quantities kept fixed) causes S to change by an amount of at most c. A statement to the effect that a change in a single random choice affects at most a certain number of output variables is to be interpreted in a similar manner.

The algorithms implied by the results listed below are needed as basic subroutines in many places. Although not all of these algorithms were originally formulated for the TOLERANT PRAM, it is not difficult to translate them to that model without loss. For all $n \in \mathbb{N}$, the *integer prefix summation problem* of size n is, given n integers $a_1, ..., a_n$, to compute the prefix sums $\sum_{j=1}^{i} a_j$, for $i = 1, ..., n$. Lemma 2.4 combines many previous results by various authors by giving the optimal prefix summation time for any combination of three independent parameters.

LEMMA 2.4 (Hagerup, 1995). *For all given integers n, m, $p \geqslant 4$, the prefix sums of n integers, each of absolute size at most m, can be computed on a* TOLERANT PRAM *using*

$$O\left(\frac{n}{p} + \frac{\log n}{\log\log p} + \log \min\left\{\frac{\log m}{\log p} + 1, n\right\}\right)$$

time, p processors and $O(n + p)$ space.

COROLLARY 2.5. *For every fixed $\delta > 0$ and for all given integers n, $\tau \geqslant 2$, the prefix sums of $(\log n)^{O(1)}$ integers, each of absolute size polynomial in n, can be computed on a* TOLERANT PRAM *using $O(\tau)$ time, $O(\lceil n^\delta/\tau \rceil)$ processors and $O(n^\delta)$ space.*

Lemmas 2.6 and 2.7 provide algorithms for the TOLERANT PRAM for problems that are trivial on certain stronger PRAM variants. Lemma 2.6 is due to Alon and Megiddo (1994), who describe a constant-time algorithm for the more general problem of linear programming in fixed dimension (to compute $\max\{a_1, ..., a_n\}$ using an algorithm for one-dimensional linear programming, minimize x subject to the constraints $x \geqslant a_i$, for $i = 1, ..., n$). Specialized to

maximum-finding, the algorithm of Alon and Megiddo can be viewed as a PRAM implementation of an algorithm for the parallel comparison-tree given by Reischuk (1985).

LEMMA 2.6 (Alon and Megiddo, 1994). *There is a constant $\varepsilon > 0$ such that for all given n, $\tau \in \mathbb{N}$, the maximum of n integers can be computed on a* TOLERANT PRAM *using $O(\tau)$ time, $\lceil n/\tau \rceil$ processors and $O(n)$ space with probability at least $1 - 2^{-n^{\varepsilon}}$ (Las Vegas).*

LEMMA 2.7 (Fich *et al.*, 1988b, Theorem 1). *For all given n, $\tau \in \mathbb{N}$, the following problem can be solved on a* TOLERANT PRAM *using $O(\tau)$ time, $\lceil n/\tau \rceil$ processors and $O(n)$ space: Given n bits $x_1, ..., x_n$, compute* $\max(\{j: 1 \leqslant j \leqslant n \text{ and } x_j = 1\} \cup \{0\})$.

When dealing with groups of consecutively numbered processors, the *segmented broadcasting* problem defined below formalizes the task of distributing information from the lowest-numbered processor in each group to the remaining group members.

DEFINITION. For all $n \in \mathbb{N}$, the *segmented broadcasting* problem of size n is, given n bits $x_1, ..., x_n$, to compute $y_1, ..., y_n$, where $y_i = \max(\{j: 1 \leqslant j < i \text{ and } x_j = 1\} \cup \{0\})$, for $i = 1, ..., n$.

Lemma 2.8 below is due to Berkman and Vishkin (1993) and Ragde (1993) (see also (Chaudhuri and Hagerup, 1994)), who in fact prove slightly stronger claims than the ones cited here.

LEMMA 2.8. *For all given $n \in \mathbb{N}$, segmented broadcasting problems of size n can be solved on a* TOLERANT PRAM

(a) *in $O(\tau)$ time using $\lceil n/\tau \rceil$ processors and $O(n)$ space, for all given integers $\tau \geqslant \log^* n$;*

(b) *in $O(\tau)$ time using $\lceil n \log^* n/\tau \rceil$ processors and $O(n \log^* n)$ space, for all given integers $\tau \geqslant 1$.*

Lemma 2.9, finally, states that small integers can be sorted fast with optimal speedup.

LEMMA 2.9. *For every fixed $\delta > 0$ and for all given integers n, m, $\tau \geqslant 4$ with $\tau \geqslant \log n/\log\log n + m^{\delta}$, n integers in the range $0..m$ can be sorted on a* TOLERANT PRAM *using $O(\tau)$ time, $\lceil n/\tau \rceil$ processors and $O(n)$ space.*

Proof. The lemma was shown by Cole and Vishkin (1989, Section 3.2) for $m \leqslant \log n/\log\log n$. Their algorithm generalizes easily to the case of larger m, with a minimum running time of $\Theta(m)$. Applying the principle of radix sort, essentially to replace m by m^{δ}, yields the bounds stated in the lemma. ∎

3. SCATTERING

The fundamental intuitive meaning of a *scattering* is that each of a number of objects is placed randomly and independently of other objects in one of a number of cells placed in a row. In this paper we are frequently interested in the resulting *fullness* of the row, i.e., in the ratio of occupied cells to the total number of cells. Since this is clearly a random variable that tends to take on larger values if more objects are scattered, it provides a (very crude) basis for estimating the number of objects scattered. By letting each object participate in the scattering with some suitable probability instead of with probability 1 as above (a *conditional scattering*), we can adjust the "region of sensitivity" of a scattering according to need. A *graduated conditional scattering* (GCS) takes this idea one step further by providing a whole sequence of conditional scatterings, each with a different associated probability of participation, which gives us a way to make more substantial statements about the number of objects scattered. Graduated conditional scatterings were introduced in (Hagerup and Radzik, 1990), although not for the purpose of estimation.

Our analysis of the outcome of a GCS is limited to determining a scattering in the sequence that satisfies a certain property, but whose successor in the sequence does not. Two properties are relevant to us: The row of a scattering being *full* (all cells are occupied), and the row being roughly half full. It turns out that testing according to full rows is computationally easier, but leads to less accurate estimates. We now provide the technical details.

DEFINITION. For all $s \in \mathbb{N}$ and $0 \leqslant p \leqslant 1$, a *conditional scattering* with probability p and of width s is a random experiment carried out by a set U as follows: Each element $u \in U$, independently of other elements, chooses a random number Z_u with $\Pr(Z_u = 0) = 1 - p$ and $\Pr(Z_u = i) = p/s$, for $i = 1, ..., s$. An element $u \in U$ is said to *occupy* the value i if $Z_u = i$, for $i = 1, ..., s$, and the *fullness* of the conditional scattering is k/s, where $k = |\{Z_u: u \in U\} \setminus \{0\}| = |\{i: 1 \leqslant i \leqslant s \text{ and } i \text{ is occupied by at least one element of } U\}|$. Two distinct elements in U *collide* if they occupy the same (nonzero) value. The *density* of the conditional scattering is the quantity $|U|\, p/s$. A scattering of width s is a conditional scattering with probability 1 and of width s.

Note that the value 0 plays a special role in the definition above. An element of U that chooses the random value 0, informally, is one that chooses not to participate in the conditional scattering; p is hence the probability of participating.

LEMMA 3.1. *Let m, $s \in \mathbb{N}$ and $0 \leqslant p \leqslant 1$, and let N be the number of occupied values in a conditional scattering with probability p and of width s carried out by a set of m elements. Then*

(a) *For all $k \in \{0, ..., s\}$,* $\Pr(N \leqslant k) \leqslant \binom{s}{k} \cdot 2^{mp(k/s - 1)}$;

(b) $\Pr(N \leqslant s/2) \leqslant 2^{s - mp/2}$;

(c) $\Pr(N < s) \leqslant s \cdot 2^{-mp/s}$;

(d) *For all $z > 0$,* $\Pr(N \geqslant z) \leqslant (mpe/z)^z$.

Proof. For all $k \in \{0, ..., s\}$, the probability that a fixed element occupies a value outside a fixed set $D \subseteq \{1, ..., s\}$ of size k is $p(s-k)/s$. Hence the probability that all occupied values belong to D is $(1-p(s-k)/s)^m \leqslant e^{-mp(s-k)/s} \leqslant 2^{mp(k/s-1)}$. Part (a) now follows by observing that D can be chosen in $\binom{s}{k}$ ways. Parts (b) and (c) are special cases of (a), and part (d) is implied by Chernoff bound (c). ∎

DEFINITION. For all $r, s \in \mathbb{N}$, a *graduated conditional scattering* (GCS) with parameters $r \times s$ of a set U is a sequence $\mathscr{S} = (\mathscr{S}_1, ..., \mathscr{S}_r)$, where $\mathscr{S}_i$, called the ith *row* of $\mathscr{S}$, is a conditional scattering with probability 2^{-i} and of width s carried out by U, for $i = 1, ..., r$. For $0 < f \leqslant 1$, define an *f-row* of $\mathscr{S}$ as any integer $i \in \{0, ..., r\}$ such that $\mathscr{S}_i$ but not $\mathscr{S}_{i+1}$ has fullness $\geqslant f$, where fictitious rows $\mathscr{S}_0$ and $\mathscr{S}_{r+1}$ are assumed to have fullness 1 and 0, respectively.

In the definition above note, in particular, that we do not require the rows of a GCS to be independent experiments; this will be crucial below. A GCS always has at least one f-row, for arbitrary f with $0 < f \leqslant 1$. Although it may have several f-rows, these will usually be sharply concentrated in a small interval. Lemma 3.3 below proves this for $f = 1$ by bounding the tail probabilities of the distribution of a 1-row L. The bounds hold no matter how L is selected from the set of 1-rows; in order for the probabilities to be well-defined, assume that this is done according to some fixed, but arbitrary rule.

LEMMA 3.2. *For every $z > 0$, $\min\{1, \sum_{i=0}^{\infty} 2^{-2^i z}\} \leqslant 2^{1-z}$.*

Proof. $\sum_{i=0}^{\infty} 2^{-2^i z} \leqslant \sum_{i=0}^{\infty} 2^{-z(i+1)} = 2^{-z}/(1-2^{-z})$. If $z \leqslant 1$, then $1 \leqslant 2^{1-z}$. On the other hand, if $z > 1$, then $2^{-z}/(1-2^{-z}) \leqslant 2^{-z}/(1/2) = 2^{1-z}$. ∎

LEMMA 3.3. *Let $m, r, s \in \mathbb{N}$ and $a > 0$ and let L be a 1-row of a GCS of a set of m elements with parameters $r \times s$. Then if $M = 2^L s$,*

(a) $\Pr(M > \max\{s, am\}) \leqslant (2e/a)^s$;

(b) $\Pr(L < r \text{ and } m > aM) \leqslant s \cdot 2^{1-a/2}$.

Proof. If $L > 0$, row L is full. Hence by Lemma 3.1(d), for every $l \geqslant 0$,

$$\Pr(L > l) \leqslant \sum_{i=\lceil l \rceil}^{\infty} \left(\frac{2^{-i} em}{s}\right)^s \leqslant \left(\frac{2^{1-l} em}{s}\right)^s. \tag{1}$$

Likewise, if $L < r$, row $L+1$ is not full, and Lemmas 3.1(c) and 3.2 imply that for every $l \leqslant r$,

$$\begin{aligned} \Pr(L < l) &\leqslant \min\left\{1, \sum_{i=-\infty}^{\lfloor l \rfloor} s \cdot 2^{\,m \cdot 2^{-i-1}/s}\right\} \\ &\leqslant s \cdot \min\left\{1, \sum_{i=0}^{\infty} 2^{-2^i m \cdot 2^{-\lfloor l \rfloor - 1}/s}\right\} \\ &\leqslant s \cdot 2^{1 - m \cdot 2^{-l-1}/s}. \end{aligned} \tag{2}$$

To show (a), apply (1) with $l = \log(\max\{s, am\}/s) \geqslant 0$. This yields

$$\Pr(M > \max\{s, am\}) = \Pr(L > l) \leqslant \left(\frac{2^{1-l} em}{s}\right)^s \leqslant (2e/a)^s.$$

To show (b), apply (2) with $l = \min\{\log(m/(as)), r\} \leqslant r$ to obtain

$$\begin{aligned} &\Pr(L < r \text{ and } m > aM) \\ &\quad = \Pr(L < l) \leqslant s \cdot 2^{1 - m \cdot 2^{-l-1}/s} \leqslant s \cdot 2^{1-a/2}. \end{aligned}$$ ∎

Lemma 3.3 provides evidence that any 1-row of a GCS of a set U can serve as a basis for estimating the size of U. We now describe how to determine such a 1-row in constant time under the assumption that each element of U has an associated processor. Carrying out the GCS in the straightforward way as a collection of independent conditional scatterings would be too slow, since each scattering would require a constant effort by each processor. Consider instead the following procedure for carrying out a GCS of U with parameters $r \times s$: Each element of U, independently of other elements, chooses a random position in an $r \times s$ matrix, the position in row i and column j being chosen with probability $2^{-i}/s$, for $i = 1, ..., r$ and $j = 1, ..., s$; with whatever probability is left (namely, 2^{-r}), the element does nothing. In more detail, the correct probability distribution among the rows can be generated by taking $i = r - \lfloor \log Z \rfloor$, where Z is a random variable uniformly distributed on the set $\{1, ..., 2^r\}$. We take the elements participating in $\mathscr{S}_i$ as precisely those that choose a position in row i of the matrix, for $i = 1, ..., r$, and the values occupied by these elements as the column numbers of their chosen matrix positions. We can now appeal to the algorithms of Lemma 3.4 below, which input the matrix positions chosen by the elements and compute from these the minimal index of a row with at least one position not chosen by any element, or $r+1$ if no such row exists; subtracting one from this number yields a 1-row of the GCS. With an eye towards later applications, we observe that the algorithms of Lemma 3.4 can also be used to analyze just a single row or a fixed selection of rows of a GCS in a similar manner, simply by ignoring the remaining rows of the matrix. Note that part (b) of Lemma 3.4 and Lemma 3.5 are needed only for the proof of Theorem 11.6, which is not part of the main development.

LEMMA 3.4. *Let U be a collection of (not necessarily distinct) elements of $\{1, ..., r\} \times \{1, ..., s\}$, for given $r, s \in \mathbb{N}$, and assume that each element of U has an associated processor. For $i = 1, ..., r$, take $f_i = 1$ if U contains (at least one occurrence of) each of the pairs $(i, 1), ..., (i, s)$, and take $f_i = 0$ otherwise. Then $\min(\{i: 1 \leqslant i \leqslant r \text{ and } f_i = 0\} \cup \{r+1\})$ can be computed in constant time*

(a) *on a* TOLERANT PRAM *using rs additional processors and O(rs) space;*

(b) *on an* ARBITRARY PRAM *using one additional processor and O(rs) space.*

Proof. (a) Let A be an $r \times s$ array and say that the cell $A[i,j]$ is *occupied* by a processor if the processor is associated with (an occurrence of) the pair (i,j), for $i = 1, ..., r$ and $j = 1, ..., s$. Then store the value 1 in each occupied cell of A and the value 0 in all remaining cells of A. On the ARBITRARY PRAM, this would be trivial; on the TOLERANT PRAM, we proceed as follows: Use the rs additional processors to associate one processor, called a *guard*, with each cell of A and let each guard begin by storing the value 1 in its associated cell. Subsequently let each processor associated with an element of U attempt to write (an arbitrary value) to the cell that it occupies; simultaneously, each guard attempts to change the value stored in its associated cell from 1 to 0. By definition of the TOLERANT PRAM, this will succeed if and only if the cell was not occupied; i.e., afterwards the cell contains the desired value. This technique, which we call *guarded writing*, was first used by Grolmusz and Ragde (1987) and appears to be a fundamental technique for programming the TOLERANT PRAM. Once the cells of A have been marked with zeros and ones as described above, it is easy to use the algorithm of Lemma 2.7 to compute the conjunction f_i of $A[i,1], ..., A[i,s]$, for $i = 1, ..., r$. Finally Lemma 2.7 is used again to determine the smallest $i \in \{1, ..., r\}$, if any, with $f_i = 0$.

(b) Our algorithm centers around a solution to a variant of a problem known as the *leftmost prisoner problem*. The leftmost prisoner problem, introduced by Fich *et al.* (1988a), is unusual in that an instance of the problem is not given by an input in a traditional sense; rather, the instance is defined by the processors available for its solution themselves. In more detail, an instance of the leftmost prisoner problem of size n is given by a set of processors numbered $1, ..., n$, each of which is either *active* or *inactive*. At least one processor is active, and the task is to compute the smallest processor number of an active processor, where inactive processors do not participate in the computation in any way. The latter restriction is essential—without it, the problem could be solved very easily using the algorithm of Lemma 2.7. The complexity of leftmost prisoner problems of size $n \geq 4$ on the ARBITRARY PRAM was shown to be $\Theta(\log \log n)$ by Chlebus *et al.* (1988) and Grolmusz (1991). Here we are interested in a variant of the problem called the *leftmost empty prison cell* problem. The setup is exactly as for the leftmost prisoner problem, but we want to compute the smallest processor number of an *inactive* processor, or an indication of the fact that all processors are active. In Lemma 3.5 below we show that leftmost empty prison cell problems can be solved in constant time on an ARBITRARY PRAM. Here we will take this result for granted and describe its application to graduated conditional scattering.

As in part (a), we use an $r \times s$ array A and say that a processor occupies $A[i,j]$ if it is associated with (i,j), for $i = 1, ..., r$ and $j = 1, ..., s$. For each row of A, we now wish to associate a processor with the row if and only if each cell in the row is occupied by at least one processor. To this end we view each row as defining an instance of the leftmost empty prison cell problem. For $j = 1, ..., s$, each processor occupying the jth cell in the row temporarily adopts j as its processor number and represents an active processor in the sense of the leftmost empty prison cell problem; the fact that several processors may occupy the same cell in A leads to no problem, since they will all carry out the same computation. For $j = 1, ..., s$, if the jth cell in the row under consideration is not occupied, we associate with it a fictitious inactive processor with processor number j. We can now use an algorithm for the leftmost empty prison cell problem to determine whether any processor is inactive, i.e., whether some cell in the row is not occupied by any processor. If and only if all cells are occupied, we associate one (or all) of the processors occupying a cell in the row with the row; note that in the special case in which no processor occupies a cell in the row, no processor will be associated with the row, as desired.

We now view the processors associated with some of the rows of A as defining an instance of the leftmost empty prison cell problem in a similar way and observe that solving this problem produces the desired result. The special case in which no row of A has an associated processor can be handled by the single processor dedicated to the GCS. ∎

LEMMA 3.5. *Leftmost empty prison cell problems of size n can be solved in constant time on an* ARBITRARY PRAM *with O(n) space.*

Proof. In the algorithm described below we shall frequently want to mark cells that we may not have been able to initialize. This is problematic, because an "undefined" value present from the outset in a cell that is not marked may happen to coincide with the value that would have been written there had the cell been marked. We avoid this difficulty by means of what we call *dynamic marking*: A processor marks a cell by first writing 0 and subsequently 1 (say) to the cell. Any processor wishing to know whether the cell is marked reads its contents both between the two writes and after the second write and deems the cell marked if and only if it observes a change from 0 to 1. Although, in this scheme, the writing and reading of a mark takes place in an interleaved fashion, in the description below we will pretend that the writing precedes the reading.

We can assume that n is a power of 2 and that at least one processor is inactive, since both requirements can be

satisfied by adding a suitable number of fictitious inactive processors (an answer larger than the number of original processors should then be interpreted as an indication that the original processors are all active). Assume that the processors are ordered linearly from left to right by increasing processor numbers. If the leftmost processor is inactive, all active processors can discover this fact through dynamic marking and output the correct answer (namely 1); assume therefore that this is not the case.

Starting from the left, divide the processors into groups of sizes 1, 1, 2, 4, ..., $n/2$ and call a group *complete* if all processors in the group are active, and *incomplete* otherwise. A first part of the computation serves to let each active processor know whether its group is complete. This can be done as follows: Using dynamic marking, each processor determines whether its left neighbor is active, where the left neighbor of the leftmost processor in a group is taken to be the rightmost processor in the group (i.e., the ordering within each group is cyclic). Then a cell associated with each group is initialized to 1 by all active processors in the group and subsequently set to 0 by all active processors in the group whose left neighbors are inactive. It is easy to see that if at least one processor in the group is active, the value of the cell remains 1 if and only if the group is complete.

The processors in incomplete groups do not participate in the remaining computation. The processors in a complete group of size m, on the other hand, use dynamic marking and the algorithm of Lemma 2.7 to solve the leftmost empty prison cell problem defined by the $m' = \min\{4m, n\}$ leftmost processors and output the result if and only if at least one of the m' leftmost processors is inactive.

Any output produced by a complete group clearly is the desired answer. On the other hand, if the processor number of the leftmost inactive processor P is k, the group to the left of P's group exists (by assumption) and is complete and of size at least $k/4$, so that an output will be produced at least by this complete group. ∎

Whereas graduated conditional scatterings were introduced for the purpose of estimation, we also employ a different kind of scattering, called *v-scattering* or (with implicit v) *multi-scattering*, for the task of placing elements in distinct cells of a destination array. Because of the more operational use, the definition below is formulated in algorithmic terms.

DEFINITION. For all $v, s \in \mathbb{N}$, to *v-scatter* a set U over an array A of s cells is to execute the following algorithm: If $v > s$, do nothing. Otherwise divide A into v disjoint subarrays of size at least $\lfloor s/v \rfloor$ each and create v copies of each element in U. Then let the set of ith copies use the ith subarray to carry out a scattering of width $\lfloor s/v \rfloor$ and identify the set of noncolliding copies, for $i = 1, \ldots, v$. An element in U is said to be *successful* if it has at least one noncolliding copy; in particular, if $v > s$ we consider all elements of U to be unsuccessful. For each successful element $u \in U$, let i and j be, respectively, the number of a noncolliding copy of u and the value occupied by that copy, and place u in the jth cell of the ith subarray of A; note that this never places distinct elements in the same cell. The *density* of the v-scattering is the quantity $|U|\, v/s$.

Using Lemma 2.7, it is easy to see that if each element of a set U has an associated group of v processors, then U can be v-scattered over an arbitrary array A in constant time. Providing each element of U with more than one copy serves to increase the probability that at least one copy will not collide (we do not care which copy this is), in which case the element can be placed in A. Lemma 3.6 quantifies the efficiency of this procedure. Its proof applies a martingale argument in a situation where, a priori, the number of random choices made is too large for such an application. We overcome this difficulty by *fixing* most of these choices in advance, i.e., by considering a restricted probability space. If we can show that some event occurs with probability at most q independently of how the random choices are fixed, then the event occurs with probability at most q even in the actual experiment, where random choices in fact are not fixed. The same principle will be used again later.

LEMMA 3.6. *Let m, s, $v \in \mathbb{N}$, denote by D the set of unsuccessful elements in a v-scattering of a set U of size m over an array of size s and let $p = mv/s$ be the density of the v-scattering. Then*

(a) *For all $u \in U$, $\Pr(u \in D) \leqslant p^v$;*

(b) *For every fixed nonempty subset R of U and for all $z \geqslant 2\,|R|\, p^v$,*

$$\Pr(|R \cap D| \geqslant z) \leqslant e^{-z^2/(32\,|R|\,v)}.$$

Proof. For part (a), it suffices to show for $v \leqslant s$ that if U carries out a scattering of width $\lfloor s/v \rfloor$, then the probability that a fixed element $u \in U$ collides is at most mv/s. If $m \geqslant s/v$, this is certainly true. Otherwise the probability under consideration is at most

$$\frac{m-1}{\lfloor s/v \rfloor} \leqslant \frac{m-1}{s/v - 1} \leqslant \frac{mv}{s}.$$

For part (b), let R be a fixed nonempty subset of U with $|R| = r$ and consider the random choices made by copies of elements not in R to be fixed in an arbitrary way. As in the proof of part (a), a fixed element in R is unsuccessful with probability at most p^v, so that $E(|R \cap D|) \leqslant rp^v$. A moment's thought reveals that a change in a single random choice (namely that of a single copy) can change $|R \cap D|$ by at most 2. Since there are altogether rv such choices, an application of Lemma 2.2(a) now shows that for $z \geqslant 2rp^v$, $|R \cap D| \geqslant z$ with probability at most $e^{-z^2/(32rv)}$. ∎

Section 6 extends the basic multi-scattering algorithm above to *colored multi-scattering*, where the set U to be multi-scattered is partitioned into *color* classes $U_1, ..., U_m$ and U_i is multi-scattered over a separate array A_i, for $i = 1, ..., m$. It is easy to see that if the density of the multi-scattering of U_i over A_i is bounded by p, for $i = 1, ..., m$, then the assertions of Lemma 3.6 carry over to the more general situation. This agrees well with intuition, since the coloring of elements only helps the multi-scattering algorithm to distribute copies evenly and avoid collisions.

4. COMPACTION

This section studies the *compaction* problem, which occurs as a base case of the more general interval allocation problem considered in Section 7. Roughly speaking, the compaction problem is to move a number of objects, scattered over a large *source* array, to distinct cells in a smaller *destination* array, possibly with a small number of objects, said to be *unlucky*, left behind in the source array. Our formalization of the problem abstracts away the identities of the objects to be moved and simply takes the input to be a sequence $x_1, ..., x_n$ of n bits, where n is the size of the source array; $x_j = 1$ signifies the presence and $x_j = 0$ the absence of an object in the jth cell of the source array, for $j = 1, ..., n$. The output takes the form of n nonnegative integers $y_1, ..., y_n$. If $x_j = 1$, the object in the jth cell of the source array can be moved to the y_jth cell of the destination array, for $j = 1, ..., n$, except that by convention $y_j = 0$ signals that the corresponding object is unlucky. If $x_j = 0$, the value of y_j is immaterial and may as well be set to zero (condition (1) below).

DEFINITION. For all $n \in \mathbb{N}$ and $s \geqslant 0$, an *incomplete placement* with *bound* s for n bits $x_1, ..., x_n$ is a sequence $y_1, ..., y_n$ of n nonnegative integers such that

(1) For $j = 1, ..., n$, if $x_j = 0$, then $y_j = 0$;

(2) For $1 \leqslant i < j \leqslant n$, if $y_i \neq 0$, then $y_i \neq y_j$;

(3) $\max\{y_j: 1 \leqslant j \leqslant n\} \leqslant s$.

The set $\{j: 1 \leqslant j \leqslant n, x_j = 1 \text{ and } y_j = 0\}$ is called the *residue set* of the incomplete placement. If the residue set is empty, the placement is said to be *complete*.

Condition (2) in the definition above expresses that distinct objects may not be placed in the same destination cell, and condition (3) states that size $\lfloor s \rfloor$ suffices for the destination array. The residue set is the set of indices of objects that are not placed in the destination array.

Most of the computational problems introduced in this paper take as (part of) their input a sequence $x_1, ..., x_n$. Although, formally, $x_1, ..., x_n$ are integers (sometimes restricted further to be single bits), informally they represent objects of additional internal structure. In particular, if $i \neq j$, the objects represented by x_i and x_j are distinct, even if it happens that $x_i = x_j$. This is mirrored closely by what happens in our algorithms for solving such problems. They typically begin by transforming the input $x_1, ..., x_n$ to n records $X_1, ..., X_n$ that are subsequently manipulated instead of $x_1, ..., x_n$. For $j = 1, ..., n$, fields in the record X_j contain the integer x_j, called the *value* of X_j, the integer j, called its *index*, as well as any other attributes that the algorithms may need. Usually we shall not describe our algorithms at the level of such programming detail; note, however, that the symbols $X_1, ..., X_n$ will be used in the sense above throughout the paper. When we speak of the jth *input element*, for $j = 1, ..., n$, we usually mean the record X_j, and $\mathcal{X} = \{X_1, ..., X_n\}$ is called the *input set*. In particular, for $i \neq j$, the ith and jth input elements are distinct.

For reasons of convenience, we will occasionally state that some algorithm is applied to a subset of $\mathcal{X}$. What we really mean in such a case is that the algorithm is applied to the corresponding subsequence of $x_1, ..., x_n$, usually permuted in some way and augmented with a number of suitable dummy elements, neither of which affects the problem in an essential way. Furthermore, we assume that enough additional information is kept to interpret the output of the algorithm in terms of the original sequence $x_1, ..., x_n$.

In the context of compaction, an *active element* is an input element of nonzero value that has not yet been placed in the destination array. Once successfully placed, we say that it has been *deactivated* or that it has become *inactive*. The incomplete compaction problem with parameters $d_1 \rightarrow_s d_2$, defined below, is, given at most d_1 active elements, to move all except at most d_2 of these to a destination array of size at most s.

DEFINITION. For all $n \in \mathbb{N}$ and $d_1, d_2, s \geqslant 0$, the *incomplete compaction* problem of size n and with parameters $d_1 \rightarrow_s d_2$ is the following: Given n bits $x_1, ..., x_n$ with $\sum_{j=1}^{n} x_j \leqslant d_1$, compute an incomplete placement for $x_1, ..., x_n$ with bound s whose residue set is of size at most d_2. If $d_2 = 0$, we speak of *complete* rather than incomplete compaction.

LEMMA 4.1. *For all given n, $d \in \mathbb{N}$, complete compaction problems of size n and with parameters $d \rightarrow_{d^4} 0$ can be solved on a (deterministic)* TOLERANT PRAM *using constant time, n processors and $O(n)$ space.*

Proof. The result was proved by Ragde for the stronger ARBITRARY PRAM (Ragde, 1993, proof of Theorem 1). Using Lemma 2.7, it is easy to translate Ragde's algorithm to the TOLERANT PRAM. ∎

Lemma 4.1 works in constant time, but places the active elements in an array with many more cells than the number of active elements. The far more important complete linear compaction problem, with inessential differences also

known as the *linear approximate compaction* or *LAC* problem (Matias and Vishkin, 1991), requires the size of the destination array to be within a constant factor of the bound on the number of active elements.

DEFINITION. For all $n \in \mathbb{N}$ and $d \geq 0$, the *complete linear compaction* problem of size n and with *limit* d is the complete compaction problem of size n and with parameters $d \to_{O(d)} 0$.

We next show that complete linear compaction problems can be solved in constant time using a superlinear number of processors. Our algorithm first multi-scatters the active elements over an auxiliary array in order to distribute them approximately evenly. The auxiliary array is then divided into segments of a fixed size chosen so large as to make it unlikely that any segment contains more than c times the average number of active elements, for a suitable constant $c > 1$. If a destination array c times larger than the number of active elements is now divided evenly among the segments, all that remains is to distribute the destination cells allocated to each segment within the segment, i.e., among the active elements stored there. This can be done via brute-force prefix summation (Corollary 2.5) following a "loose" compaction of the active elements within the segment (Lemma 4.1). The details follow.

LEMMA 4.2. *For every fixed $\delta > 0$, there is a constant $\varepsilon > 0$ such that for all given n, $d \in \mathbb{N}$, complete linear compaction problems of size n and with limit d can be solved on a* TOLERANT PRAM *using constant time, $O(n^{1+\delta})$ processors and $O(n^{1+\delta})$ space with probability at least $1 - 2^{-n^{\varepsilon}}$ (Las Vegas).*

Proof. Let $v = \lceil 8/\delta \rceil$. Without loss of generality we can assume that δ is rational (so that we can easily compute with δ; cf. Section 2) and that $n \geq 4$ and $d \leq n/v$ (since otherwise the compaction problem is trivial). It suffices to describe an algorithm that uses constant time, $O(n^{1+\delta/2})$ processors and $O(n^{1+\delta/2})$ space and that fails with probability at most $1/2$, since we can execute such an algorithm $n^{\Omega(1)}$ times in parallel and select as our output the outcome of any successful execution. If $d < \log n$, the problem can be solved by first using the algorithm of Lemma 4.1 to move the active elements to an array of $(\log n)^{O(1)}$ cells and subsequently compacting them exactly, i.e., numbering them consecutively, using prefix summation (Corollary 2.5). Assume hence that $d \geq \log n$.

Let A be an array of size vs, where s is chosen as a multiple of $r = \lceil d/\lfloor \log n \rfloor \rceil$ with $s \geq n^{1+\delta/4}$, but $s = O(n^{1+\delta/4})$. Then v-scatter the active elements in the source array over A; by Lemma 3.6(a), the probability that some element cannot be placed in A is at most $n \cdot (d/s)^v \leq n \cdot (n^{-\delta/4})^{8/\delta} = 1/n$.

Divide A into vr disjoint *segments* of size s/r each. The number S of active elements placed in a fixed segment in the v-scattering above is clearly bounded by the number S' of copies of elements choosing a cell in the segment in the v-scattering (S may be smaller than S' because copies choosing a cell in the segment can collide, and still smaller because elements with a noncolliding copy placed in the segment may be moved to the position of another noncolliding copy). Since the v-scattering partitions A into v subarrays of r segments each and at most d copies choose cells in the subarray containing the segment under consideration, S' is binomially distributed with expected value at most $d/r \leq \log n$; Chernoff bound (a) therefore implies that $S' \geq 12 \log n$ with probability at most $e^{-2 \log n} \leq n^{-2}$. It follows that except with probability at most $n \cdot n^{-2} = 1/n$, no segment contains more than $12 \log n$ active elements.

Since we have $n^{\Omega(1)}$ processors per segment, we can now use the algorithms of Lemma 4.1 and Corollary 2.5 as in the beginning of the proof to attempt to place the active elements in each segment in an array of size $12\lceil \log n \rceil$ (the attempt fails only in the unlikely event that some segment contains more than $12\lceil \log n \rceil$ elements). Assigning to each segment a subarray of size $12\lceil \log n \rceil$ of a common destination array and moving each active element to the appropriate cell in the destination array completes the compaction. The total size of the destination array is $12vr\lceil \log n \rceil = O(d)$, and the probability that the algorithm fails is at most $2/n \leq 1/2$. ∎

COROLLARY 4.3. *For every fixed $\delta > 0$ there is a constant $\varepsilon > 0$ such that for all given n, d, $\tau \in \mathbb{N}$, complete linear compaction problems of size n and with limit $d = O(n^{1-\delta})$ can be solved on a* TOLERANT PRAM *using $O(\tau)$ time, $\lceil n/\tau \rceil$ processors and $O(n)$ space with probability at least $1 - 2^{-n^{\varepsilon}}$ (Las Vegas).*

Proof. Without loss of generality assume that δ is rational and that $\delta \leq 1$. Since the problem is easily solved using standard prefix summation (Lemma 2.4) if $\tau = n^{\Omega(1)}$, we can further assume the availability of at least $n^{1-\delta/4}$ processors. Divide the input set $\mathcal{X}$ into $O(n^{1-\delta/2})$ *clusters* of $O(n^{\delta/2})$ input elements each and call a cluster *nonempty* if it contains at least one active element. There are obviously at most d nonempty clusters, so the algorithm of Lemma 4.2 can be used to place the indices of these in an array of size $O(d)$. This implicitly places all active elements in an array of size $O(d \cdot n^{\delta/2}) = O(n^{1-\delta/2})$, after which the compaction can be completed via a second application of the algorithm of Lemma 4.2. ∎

Matias and Vishkin (1991) showed that complete linear compaction problems of size n and with limit d can be solved in $O(\log^* d)$ time with n processors by successively solving $O(\log^* d)$ incomplete compaction problems. The basic idea is that the gradual deactivation of elements frees resources that can be used to speed up the rate of deactivation, thus leading to the fast convergence of the algorithm. In more detail, Matias and Vishkin show that if the number

of active elements has already dropped to d/v^c, for a suitable constant $c>0$ and for some $v \in \mathbb{N}$, then in constant time it can be decreased further to $d/2^{cv}$. The algorithm of (Matias and Vishkin, 1991) realizing this claim is reasonably complicated and relies crucially on Lemma 4.1. We give a simple algorithm for the same task whose use of Lemma 4.1, although convenient for the exposition, is inessential and can be avoided, and whose complete analysis is much simpler than what would be required for the algorithm of Matias and Vishkin. As concerns the claim of simplicity, observe below that the appeal to Corollary 4.3 is needed only to deal with a special case that was not even considered in (Matias and Vishkin, 1991).

Our algorithm for incomplete compaction inputs at most d/v^3 active elements stored in a source array of size n and places all except $d/2^{3v}$ of these in a destination array of size $O(d/v)$. The basic idea is to $5v$-scatter the active elements over an array of size $10d/v^2$. Since, by assumption, the density of this $5v$-scattering is at most $1/2$, a fixed active element remains active with probability at most 2^{-5v} (Lemma 3.6(a)), which allows us to conclude that with high probability the size of the residue set will be bounded by $d/2^{3v}$. The only problem with this approach is that we do not know how to allocate the $5v$ processors per active element necessary to carry out the $5v$-scattering in constant time. Similarly as in the proof of Corollary 4.3, we therefore divide the input set $\mathscr{X}$ into *clusters* of size v each and $5v$-scatter not the active elements themselves, but instead (the indices of) the *nonempty* clusters, i.e., those clusters that contain at least one active element. Since the number of nonempty clusters is obviously bounded by the number of active elements, the density of the modified $5v$-scattering is also at most $1/2$. Furthermore, the allocation of $5v$ processors to each cluster is trivial, and placing the nonempty clusters in an array of size $O(d/v^2)$ implicitly places the active elements in an array of size $O(d/v)$.

LEMMA 4.4. *There is a constant $\varepsilon>0$ such that for all given $n, d, v, \tau \in \mathbb{N}$, incomplete compaction problems of size n and with parameters*

$$\frac{d}{v^3} \xrightarrow{O(d/v)} \frac{d}{2^{3v}}$$

can be solved on a TOLERANT PRAM *using $O(\tau)$ time, $\lceil n/\tau \rceil$ processors and $O(n)$ space with probability at least $1-2^{-n^{\varepsilon}}$ (Monte Carlo).*

Proof. We give the proof for $\tau=1$, leaving the easy extension to general values of τ to the reader (informally, the observation needed is that executing a multi-scattering over several steps rather than in one step can only cause more elements to be successful). We can obviously assume that $v^3 \leqslant d$ (otherwise we start with no active elements), that $d \leqslant nv$ (otherwise the trivial placement with bound n will do), and therefore that $d \leqslant n^{3/2}$.

Consider the following algorithm:

Step 1. Divide $\mathscr{X}$ into $l=\lceil n/v \rceil$ *clusters* $\mathscr{X}_1, \ldots, \mathscr{X}_l$ of at most v input elements each and use the algorithm of Lemma 2.7 to compute a bit vector representation of the set $I=\{i: 1 \leqslant i \leqslant l \text{ and } \mathscr{X}_i \text{ contains at least one active element}\}$.

Step 2. Associate $5v$ processors with each element of I and $5v$-scatter I over an array of size $\lceil 10d/v^2 \rceil$; let $I' \subseteq I$ denote the set of unsuccessful indices. Use the outcome of the $5v$-scattering to place all active elements in $\bigcup_{i \in I \setminus I'} \mathscr{X}_i$ in an array of size $v\lceil 10d/v^2 \rceil = O(d/v)$.

The algorithm clearly runs on a TOLERANT PRAM within the desired resource bounds. A fixed active element remains active exactly if the index of its cluster is unsuccessful in the $5v$-scattering in Step 2. By Lemma 3.6(b), the number of such unsuccessful cluster indices is bounded by $\max\{2d/2^{5v}, n^{5/6}/v\}$, except with probability at most $e^{-\zeta}$, where $\zeta=(n^{5/6}/v)^2/(32(d/v^3)\cdot 5v)=\Omega(n^{5/3}/d)=\Omega(n^{1/6})$. With high probability the number of active elements therefore decreases to at most $v\cdot\max\{d/2^{4v}, n^{5/6}/v\} \leqslant \max\{d/2^{3v}, n^{5/6}\}$. If this is more than $d/2^{3v}$, at most $n^{5/6}$ elements remain active, and these can be deactivated via an application of the algorithm of Corollary 4.3. ∎

COROLLARY 4.5. *There is a constant $\varepsilon>0$ such that for all given n, $d \in \mathbb{N}$, complete linear compaction problems of size n and with limit d can be solved on a* TOLERANT PRAM *using $O(\log^* d)$ time, n processors and $O(n)$ space with probability at least $1-2^{-n^{\varepsilon}}$ (Las Vegas).*

Proof. Assume that $d \leqslant n$ and apply the algorithm of Lemma 4.4 at most $\log^* d$ times in successive *stages*, starting with $v=1$. Each stage after the first attempts to place the unlucky elements of the previous stage in a new but smaller array. Schematically,

$$\frac{d}{1^3} \xrightarrow{O(d/1)} \frac{d}{2^3} \xrightarrow{O(d/2)} \frac{d}{4^3} \xrightarrow{O(d/4)} \frac{d}{16^3} \xrightarrow{O(d/16)} \frac{d}{(2^{16})^3} \to \cdots \to 0.$$

The total size of the destination arrays is

$$O(d(1+\tfrac{1}{2}+\tfrac{1}{4}+\tfrac{1}{16}+\cdots))=O(d). \quad ∎$$

As mentioned above, a weaker form of Corollary 4.5 was first proved by Matias and Vishkin (1991), who also noted that it has applications to processor scheduling as per Brent's principle. We next describe an improved algorithm that achieves optimal speedup. A similar result was derived in a somewhat different way by Goodrich (1991), and a slightly slower algorithm with optimal speedup was described previously by Matias and Vishkin (1991).

THEOREM 4.6. *There is a constant $\varepsilon > 0$ such that for all given n, d, $\tau \in \mathbb{N}$ with $\tau \geqslant \log^* d$, complete linear compaction problems of size n and with limit d can be solved on a* TOLERANT PRAM *using $O(\tau)$ time, $\lceil n/\tau \rceil$ processors and $O(n)$ space with probability at least $1 - 2^{-n^{\varepsilon}}$ (Las Vegas).*

Proof. Assume that $\tau \leqslant (\log n)/32$, since otherwise the compaction can be carried out using prefix summation (Lemma 2.4), and that $d \leqslant n$. We describe a preprocessing stage that reduces the problem size from n to $O(n/\tau)$. Divide $\mathscr{X}$ into $\lceil n/\tau \rceil$ *clusters* of at most τ input elements each and associate a processor with each cluster. Using a global array A of size $8d$, the processors now execute 2τ *rounds*. In each round, each processor chooses an active element in its cluster, if any are left, and attempts to place the chosen element in a random cell of A. If the cell is not already occupied and there is no collision, the element is placed and becomes inactive. It is easy to see that each such trial fails with probability at most $1/8$, even conditionally on any pattern of failures in previous rounds. As a consequence, all of τ fixed trials by a fixed processor fail with probability at most $(1/8)^{\tau}$ (if the processor runs out of active elements, let it subsequently execute dummy trials that always succeed). However, if a fixed processor has any active elements left after 2τ rounds (call such a processor *busy*), at least τ of its trials must have failed, which, by the above, happens with probability at most $\binom{2\tau}{\tau}(1/8)^{\tau} \leqslant 2^{2\tau} \cdot 2^{-3\tau} = 2^{-\tau}$. The expected number of busy processors therefore is $O(n/2^{\tau})$. Our intent is to use a martingale argument to show that with high probability, the actual number of busy processors is $O(n/\tau^2)$, which requires us to bound the effect on the number of busy processors of a change in a single random choice. A change in a single random choice here is the choice by some processor of a different cell in A in some round. Say that a processor P is *affected* (by the change under consideration) in a given round if the change influences the success of P's trial in the given round or in some earlier round. At most two processors are affected in the first round, and it is easy to see that the number of affected processors at most triples from one round to the next—an affected processor can "affect" at most two other processors in each later round. Therefore the total number of affected processors after 2τ rounds is at most $3^{2\tau} \leqslant 2^{4\tau} \leqslant n^{1/8}$; this is an upper bound on the change in the number of busy processors caused by a change in a single random choice. Since the algorithm makes a total of $O(n)$ random choices, a martingale argument now shows that with high probability, the actual number of busy processors is $O(n/2^{\tau} + n^{1/8} \cdot n^{5/8}) = O(n/\tau^2)$. But then the algorithm of Corollary 4.5 can be used to place (the processor numbers of) the busy processors in an array of size $O(n/\tau^2)$. This implicitly places the remaining active elements in an array of size $O(n/\tau)$, and the compaction can be completed via another application of the algorithm of Corollary 4.5. ∎

5. FINE-PROFILING

The present paper studies several different kinds of profiling problems. In general terms, the task is, given an array containing occurrences of several different values, to estimate the *multiplicity* of each value, i.e., the number of occurrences of that value. Estimation procedures somewhat similar to our fine-profiling algorithm in the present section were developed in independent work by Gil *et al.* (1991) and Goodrich (1991), while a profiling algorithm of a flavor similar to that of our coarse-profiling algorithm in Section 8 was described in (Bast and Hagerup, 1991).

We now introduce convenient notation and terminology that will be used throughout the remainder of the paper. In the context of an input consisting of n integers $x_1, \ldots, x_n$ in the range $0..m$, for $n, m \in \mathbb{N}$, we take $\mathscr{B}_i = \{X_j : 1 \leqslant j \leqslant n \text{ and } x_j = i\}$ and $b_i = |\mathscr{B}_i|$, for $i = 1, \ldots, m$. For $i = 1, \ldots, m$, the integer i will also be called a *color*, $\mathscr{B}_i$ is a *color class*, and b_i is called the *multiplicity* of the color i. For $1 \leqslant i < j \leqslant m$, we consider $\mathscr{B}_i$ and $\mathscr{B}_j$ to be distinct even if they happen to contain the same elements (this is possible only if $\mathscr{B}_i = \mathscr{B}_j = \varnothing$). When a color class $\mathscr{B}_i$ is manipulated as a single object by some algorithm, it is represented by its index i. Note that elements of value 0 are not considered to belong to any color class. They are just "dummy elements" that represent the absence of a true element. Whenever we have dealt with certain color classes in some algorithm, we can "remove" the elements of these color classes by setting their values to 0, which allows us to focus on the remaining color classes. This will be used on several occasions.

Given n integers $x_1, \ldots, x_n$ in the range $0..m$, for $n, m \in \mathbb{N}$, an *m-color profile* for $x_1, \ldots, x_n$ is a sequence $\hat{b}_1, \ldots, \hat{b}_m$ of m nonnegative integer random variables, the idea being that $\hat{b}_i$ is an estimate of b_i, for $i = 1, \ldots, m$. A *fine-profile*, defined below, provides estimates that are correct up to a constant factor; for reasons of convenience we also require each estimate to be no smaller than the true multiplicity. For our purposes, having such estimates usually is as good as knowing the exact multiplicities.

DEFINITION. Let $n, m \in \mathbb{N}$ and let $x_1, \ldots, x_n$ be n integers in the range $0..m$. For $i = 1, \ldots, m$, take $b_i = |\{j : 1 \leqslant j \leqslant n \text{ and } x_j = i\}|$. An *$m$-color fine-profile* for $x_1, \ldots, x_n$ is a sequence $\hat{b}_1, \ldots, \hat{b}_m$ of m nonnegative integer random variables such that $b_i \leqslant \hat{b}_i \leqslant Kb_i$, for $i = 1, \ldots, m$ and for some constant $K \geqslant 1$. If additionally $\hat{b}_1, \ldots, \hat{b}_m$ are independent, the sequence $\hat{b}_1, \ldots, \hat{b}_m$ is called a *strong fine-profile* for $x_1, \ldots, x_n$. The m-color (strong) fine-profiling problem of size n is, given n and m, to compute an m-color fine-profile (composed of independent estimates) for n given integers in the range $0..m$.

We define a *linear overestimate* for a quantity b as a random variable $\hat{b}$ with $b \leqslant \hat{b} \leqslant Kb$, for some constant $K \geqslant 1$. An m-color fine-profile for $x_1, \ldots, x_n$ may therefore also be

characterized as a sequence of linear overestimates for $b_1, ..., b_m$, all with the same implicit constant K.

A statement quite similar to Lemma 5.1 below can be derived by combining results of (Stockmeyer, 1983) and (Ajtai and Ben-Or, 1984) with the obvious simulation of unbounded fan-in circuits by CRCW PRAMs. We give a somewhat different proof, which in the context of PRAMs seems more direct.

LEMMA 5.1. *For every fixed $\delta > 0$ there is a constant $\varepsilon > 0$ such that for all given $n \in \mathbb{N}$, the following problem can be solved on a* TOLERANT PRAM *using constant time, $O(n^\delta)$ processors and $O(n)$ space with probability at least $1 - 2^{-n^\varepsilon}$ (Monte Carlo): Given n bits $x_1, ..., x_n$, compute a bit y such that*

(1) $\sum_{j=1}^{n} x_j \geq n/2 \Rightarrow y = 1$;

(2) $\sum_{j=1}^{n} x_j \leq n/8 \Rightarrow y = 0$.

Proof. The idea of the proof, which the reader may appreciate better after the first reading, is to "amplify" a constant-factor difference to a "polynomial" difference, which can then easily be detected using Ragde's lemma (Lemma 4.1).

Assume that δ is rational, that $\delta \leq 1$ and that $n \geq 16$, take $h = 4\lfloor n^{\delta/2} \rfloor \leq n$ and let $t = 32\lceil \log n \rceil$. Begin by determining the number of ones in each of h random samples of t input bits each; i.e., choose ht independent random numbers $z_{1,1}, ..., z_{1,t}, z_{2,1}, ..., z_{2,t}, ..., z_{h,1}, ..., z_{h,t}$ from the uniform distribution over $\{1, ..., n\}$ and use the algorithm of Corollary 2.5 to compute $S_i = \sum_{j=1}^{t} x_{z_{i,j}}$, for $i = 1, ..., h$. The random variables $S_1, ..., S_h$ are independent and binomially distributed with expected value tb/n, where $b = \sum_{j=1}^{n} x_j$. Hence by Lemma 2.1, the following holds for $i = 1, ..., h$: If $b \geq n/2$, then $\Pr(S_i \leq 8\lceil \log n \rceil) \leq n^{-1}$, while if $b \leq n/8$, then $\Pr(S_i > 8\lceil \log n \rceil) \leq n^{-1}$. For $i = 1, ..., h$, take $A[i] = 1$ if $S_i > 8\lceil \log n \rceil$, and let $A[i] = 0$ otherwise. The remaining problem is, assuming that the vast majority of $A[1], ..., A[h]$ has a common value (0 or 1), to find that value. Do this by attempting, using the algorithm of Lemma 4.1 with $d = \lfloor (h/4)^{1/4} \rfloor$, to move the set of ones in A to an array of size $h/4$. Set $y = 1$ if and only if this fails.

In order to analyze the last part of the algorithm, note that $S = \sum_{i=1}^{h} A[i]$ is binomially distributed, and that the preceding discussion implies that $E(S) \geq h/2$ if $b \geq n/2$, while $E(S) \leq 1$ if $b \leq n/8$. By another application of Lemma 2.1, the following happens with high probability: $S > h/4$ if $b \geq n/2$, while $S \leq (h/4)^{1/4}$ if $b \leq n/8$. In the first case, the compaction using the algorithm of Lemma 4.1 surely fails, while in the second case it will succeed. In either case y receives the correct value. ∎

When using the algorithm of Lemma 5.1 to analyze the outcome of a GCS $\mathscr{S} = \{\mathscr{S}_1, ..., \mathscr{S}_r\}$ below, we apply the algorithm separately to each row of $\mathscr{S}$ and define a row to be *almost-full* if the algorithm assigns the value 1 to the bit y associated with the row. The *threshold* of $\mathscr{S}$ is 0 if $\mathscr{S}_1$ is not almost-full, and otherwise is the largest integer $i \in \{1, ..., r\}$ such that $\mathscr{S}_j$ is almost-full, for $j = 1, ..., i$.

In analogy with the definition of an f-row of a GCS and motivated by Lemma 5.1, define an *(f_1, f_2)-row* of a GCS $\mathscr{S} = (\mathscr{S}_1, ..., \mathscr{S}_r)$, for $0 \leq f_1 < f_2 \leq 1$, as any integer $i \in \{0, ..., r\}$ such that $\mathscr{S}_i$ has fullness $> f_1$ and $\mathscr{S}_{i+1}$ has fullness $< f_2$, where fictitious rows $\mathscr{S}_0$ and $\mathscr{S}_{r+1}$ are assumed to have fullness 1 and 0, respectively. Lemma 5.1 shows that the threshold of a GCS with high probability is a $(\frac{1}{8}, \frac{1}{2})$-row of the GCS.

LEMMA 5.2. *Let $m, r, s \in \mathbb{N}$ and let L be a $(\frac{1}{8}, \frac{1}{2})$-row of a GCS of a set of m elements with parameters $r \times s$. Let $M = 2^L s$ and take $c_1 = 1/(2^{12}e)$ and $c_2 = 12$. Then*

(a) *If $m \geq c_1 s$, then $\Pr(m < c_1 M) \leq 2^{-s}$;*

(b) *$\Pr(L > 0$ and $m < c_1 M) \leq 2^{-s}$;*

(c) *If $r \geq \lfloor \log m \rfloor$, then $\Pr(m > c_2 M) \leq 2^{-s}$.*

Proof. We proceed as in the proof of Lemma 3.3. If $L > 0$, the fullness of row L is at least $\frac{1}{8}$. Hence by Lemma 3.1(d), for every $l \geq 0$,

$$\Pr(L > l) \leq \sum_{i=\lceil l \rceil}^{\infty} \left(\frac{8em \cdot 2^{-i}}{s}\right)^{\lceil s/8 \rceil} \leq 2\left(\frac{8em \cdot 2^{-l}}{s}\right)^{\lceil s/8 \rceil} \leq \left(\frac{2^{4-l}em}{s}\right)^{\lceil s/8 \rceil}. \tag{3}$$

Likewise, if $L < r$, the fullness of row $L + 1$ is at most $1/2$. Hence by Lemmas 3.1(b) and 3.2, for every $l \leq r$,

$$\begin{aligned}\Pr(L < l) &\leq \min\left\{1, \sum_{i=-\infty}^{\lfloor l \rfloor} 2^{s - m \cdot 2^{-l-2}}\right\} \\ &\leq 2^s \cdot \min\left\{1, \sum_{i=0}^{\infty} 2^{-2^{l} m \cdot 2^{-\lfloor l \rfloor - 2}}\right\} \\ &\leq 2^{s+1-m \cdot 2^{-l-2}}.\end{aligned} \tag{4}$$

To verify (a), apply (3) with $l = \log(m/(c_1 s)) \geq 0$ to obtain

$$\Pr(m < c_1 M) = \Pr(L > l) \leq \left(\frac{2^{4-l}em}{s}\right)^{\lceil s/8 \rceil} \leq 2^{-s}.$$

(b) follows immediately from (a) and the observation that $L > 0$ implies $m \geq s/8 \geq c_1 s$. To verify (c), apply (4) with $l = \log(m/(c_2 s)) \leq r$ to obtain

$$\Pr(m > c_2 M) = \Pr(L < l) \leq 2^{s+1-m \cdot 2^{-l-2}} = 2^{s+1-c_2 s/4} = 2^{1-2s} \leq 2^{-s}. \quad ∎$$

The algorithm described in the theorem below outputs a sequence of independent integers, except that it may fail and not produce any output at all. As regards the independence, the precise statement is that for each input and conditionally on the event that any output is produced, the integers output by the algorithm are independent. Similar interpretations should be imposed on other results in the sequel concerning randomized algorithms that are claimed to output independent random numbers. A simpler proof of a statement similar to Theorem 5.3 was indicated by Goodrich (1991).

THEOREM 5.3. *For every fixed $\delta > 0$ there is a constant $\varepsilon > 0$ such that for all given n, m, $\tau \in \mathbb{N}$ with $m = O(n^{1-\delta})$, m-color strong fine-profiling problems of size n can be solved on a* TOLERANT PRAM *using $O(\tau)$ time, $\lceil n/\tau \rceil$ processors and $O(n)$ space with probability at least $1 - 2^{-n^{\varepsilon}}$ (Monte Carlo).*

Proof. The idea of the algorithm is simple: If the size of a color class is $n^{\Omega(1)}$, it can be reliably estimated using a GCS; otherwise the color class can be blown up by a factor of $n^{\Theta(1)}$ and its size estimated in the same way. We now provide the details.

Without loss of generality assume that δ is rational and at most 1 and that $n \geq 2$. Since computing exact multiplicities reduces to sorting, the given problem is easily solved using the algorithm of Lemma 2.9 if $\tau = n^{\Omega(1)}$; we can therefore also assume the availability of at least $n^{1-\delta/3}$ processors. Let $s = \lceil n^{\delta/3} \rceil$. For $i = 1, \ldots, m$, use guarded writing to set $\hat{b}_i = 0$ if $b_i = 0$, and otherwise carry out the following procedure, where c_1 and c_2 are as defined in Lemma 5.2:

Step 1. Using the algorithm of Lemma 5.1, execute a GCS $\mathscr{S}_i$ of $\mathscr{B}_i$ with parameters $\lfloor \log n \rfloor \times s$ and compute l_i as the threshold of $\mathscr{S}_i$. Note that we need a guard processor for each cell of each GCS, and that by assumption sufficiently many processors are available.

Step 2. If $l_i > 0$, take $\hat{b}_i = c_2 \cdot 2^{l_i} s$. Otherwise use the algorithm of Corollary 4.3 to allocate s processors to each element of $\mathscr{B}_i$, let these processors execute a GCS $\mathscr{S}'_i$ with parameters $\lfloor \log (sn) \rfloor \times s$ and take $\hat{b}_i = c_2 \cdot 2^{l'_i}$, where l'_i is the threshold of $\mathscr{S}'_i$.

It is easy to see that the space needed by the algorithm is $O(n)$. To see that the same holds for the number of operations, note that by Lemma 5.2(c), no processors are allocated to a fixed color class $\mathscr{B}_i$ with $b_i > c_2 s$, except with negligible probability.

Now fix $i \in \{1, \ldots, m\}$. The following happens with high probability: If $l_i > 0$, we have $b_i \leq \hat{b}_i \leq (c_2/c_1)\, b_i$ (by Lemma 5.2, parts (b) and (c)). If $l_i = 0$, then $sb_i \leq c_2 \cdot 2^{l'_i} s \leq (c_2/c_1)\, sb_i$; i.e., $b_i \leq \hat{b}_i \leq (c_2/c_1)\, b_i$ (by Lemma 5.2, parts (a) and (c)). In either case, $\hat{b}_i$ is a linear overestimate for b_i.

The estimates produced by the algorithm are clearly independent unless the processor allocation according to Corollary 4.3 fails, in which case the algorithm can report failure and refrain from producing any output. ∎

Remark. The fine-profiling algorithm above is Monte Carlo; i.e., we cannot detect if the estimates computed are off by more than the allowed constant factor. In Section 10 we derive a Las Vegas fine-profiling algorithm (Corollary 10.5).

6. COLORED COMPACTION

It is essential for the application to interval allocation described in Section 7 as well as for other reasons to generalize the compaction problem studied in Section 4 to *colored compaction*, where objects of different *colors*, initially placed in a single source array, are to be moved to distinct destination arrays, one for each color. As before, an object that cannot be placed is called unlucky, and we will not distinguish between an object and the input element representing it. In the formal definition below, the value of an element represents its color, the special value 0 still representing the absence of an object. This is in agreement with our terminology concerning $\mathscr{B}_1, \ldots, \mathscr{B}_m$.

DEFINITION. Given $n, m \in \mathbb{N}$ and $d_1, \ldots, d_m \geq 0$ as well as n integers $x_1, \ldots, x_n$ in the range $0..m$, an *incomplete placement* for $x_1, \ldots, x_n$ with bounds $d_1, \ldots, d_m$ is a sequence $y_1, \ldots, y_n$ of n nonnegative integers such that

(1) For $j = 1, \ldots, n$, if $x_j = 0$, then $y_j = 0$;

(2) For $1 \leq i < j \leq n$, if $x_i = x_j$ and $y_i \neq 0$, then $y_i \neq y_j$;

(3) For $i = 1, \ldots, m$, $\max(\{y_j\colon 1 \leq j \leq n \text{ and } x_j = i\} \cup \{0\}) \leq d_i$.

The set $\{j\colon 1 \leq j \leq n, x_j \neq 0 \text{ and } y_j = 0\}$ is called the *residue set* of the incomplete placement. If the residue set is empty, the placement is *complete*.

Condition (2) ensures that distinct elements of the same color are not placed in the same destination cell, and condition (3) states that the lucky elements of $\mathscr{B}_i$ fit into an array of size $\lfloor d_i \rfloor$, for $i = 1, \ldots, m$. If there is only a single color, i.e., for $m = 1$, the definition above reduces to our earlier definition of an incomplete placement.

The compaction problems introduced in Section 4 generalize in a natural way to colored compaction. Our next goal is to extend the compaction algorithms given for the special case of a single color to the case of several colors. Recall that to a first approximation, the algorithm of Lemma 4.4 multi-scatters the active elements over an array of size s, for suitably chosen s. A straightforward generalization to the case of m colors would be to multi-scatter the elements of $\mathscr{B}_i$ over an array associated with $\mathscr{B}_i$ and of a suitably chosen size s_i, for $i = 1, \ldots, m$. Attempting this, we are faced with a somewhat extraneous problem, namely that we do not know how to allocate m disjoint arrays of

prescribed sizes $s_1, ..., s_m$ sufficiently fast without wasting too much space. Lemma 6.1 below therefore assumes "pre-allocated" such arrays to be supplied by any "user" of the lemma; the sizes of these arrays simultaneously serve as the bounds $d_1, ..., d_m$. As is rather obvious, the compaction of a particular color class will not be very successful unless the size of the array provided for that color class is considerably larger than the size of the color class—we later define such color classes to be *well-supplied*. Lemma 6.1 therefore identifies the set of (indices of) elements in well-supplied color classes, and its assertions apply only to such elements.

A more interesting complication in the generalization of the algorithm of Lemma 4.4 to the case of several colors lies in the fact that in Step 2 of the algorithm, several elements forming a cluster are multi-scattered together. While this works fine in the case of a single color, it is not appropriate if the elements in a cluster have different colors, i.e., are to be placed in different destination arrays. In order to solve this problem, recall that the clusterwise scattering was motivated by efficiency considerations: multi-scattering single elements works just as well, but requires several processors standing by each active element. The idea now is first to compact the active elements as though they were all of the same color—we already know how to do that—but to use the outcome of this compaction exclusively to allocate the necessary processors to each active element, after which the colored compaction can be completed in the simple way described above. As suggested by this description, an initial part of our algorithm for colored compaction is quite similar to the corresponding algorithm for (uncolored) compaction. Because of the slight differences and since we want to extend the analysis of the algorithm given earlier, however, we essentially reproduce it as Steps 1 and 2 of the algorithm of Lemma 6.1 below.

Before we state the lemma, recall that for $c > 0$, a real-valued function S defined on a set M equipped with a metric ϕ is said to satisfy a *Lipschitz condition* with constant c if for all $x, y \in M$, we have $|S(x) - S(y)| \leqslant c \cdot \phi(x, y)$. The only metric space relevant to the present paper, and the one implicitly intended in every reference to a Lipschitz condition, is the set of subsets of $\{1, ..., n\}$, equipped with the metric ϕ with $\phi(U, V) = |U \triangle V|$, for all $U, V \subseteq \{1, ..., n\}$, where $U \triangle V$ denotes the symmetric difference of U and V; i.e., $U \triangle V = (U \setminus V) \cup (V \setminus U)$.

LEMMA 6.1. *There is a constant $\varepsilon > 0$ such that the following holds: Let $n, m, v, \tau \in \mathbb{N}$ be given, suppose that $x_1, ..., x_n$ are n given integers in the range $0..m$ and let $A_1, ..., A_m$ be m given nonoverlapping arrays. Take $B_i = \{j: 1 \leqslant j \leqslant n \text{ and } x_j = i\}$ and $b_i = |B_i|$, for $i = 1, ..., n$, and define $J = \{i: 1 \leqslant i \leqslant m \text{ and } |A_i| \geqslant 6vb_i\}$, $B' = \bigcup_{i \in J} B_i$ and $b = \sum_{i=1}^{m} b_i$. Then an incomplete placement for $x_1, ..., x_n$ with bounds $|A_1|, ..., |A_m|$ can be computed on a* TOLERANT PRAM *using $O(\tau)$ time, $\lceil (n + v^3 b)/\tau \rceil$ processors and $O(n + v^3 b)$ space with probability at least $1 - 2^{-n^{\varepsilon}}$ (Monte Carlo), such that the residue set D of the placement satisfies conditions* (a)–(c):

(a) *For every $j \in B'$, $\Pr(j \in D) \leqslant 2^{-\tau}$;*

(b) *For every fixed nonempty subset R of B' and for all $z \geqslant |R|/2^{\tau}$,*

$$\Pr(|R \cap D| \geqslant z) \leqslant 2e^{-z^2/(2^9 |R| v^3)};$$

(c) *For every nonnegative real function S of D that satisfies a Lipschitz condition with constant c,*

$$S = O(E(S) + cv^3 n^{5/8})$$

with probability at least $1 - 2^{-n^{1/8}}$.

Remark. As anticipated above, a color class $\mathscr{B}_i$ is called *well-supplied*, in the context of Lemma 6.1, if $|A_i| \geqslant 6vb_i$, so that B' is the set of indices of elements in well-supplied color classes. Part (a) of the lemma says that any fixed element of a well-supplied color class is unlikely to remain active, part (b) extends this property from single elements to arbitrary sets of elements in well-supplied color classes, and part (c) states that any function of the residue set that satisfies a Lipschitz condition with a constant that is not too large with high probability does not significantly exceed its expected value.

Proof. With the same justification as in the proof of Lemma 4.4, we give the proof only for $\tau = 1$. Start by using the algorithm of Theorem 5.3 to compute an estimate $\hat{b}$ of b and assume that $\hat{b}$ is indeed a linear overestimate for b. Then execute the following:

Step 1. Divide $\mathscr{X}$ into $l = \lceil n/v \rceil$ *clusters* $\mathscr{X}_1, ..., \mathscr{X}_l$ of at most v input elements each and use the algorithm of Lemma 2.7 to compute a bit vector representation of the set $I = \{j: 1 \leqslant j \leqslant l \text{ and } \mathscr{X}_j \text{ contains at least one active element}\}$.

Step 2. Associate $4v$ processors with each element of I and $4v$-scatter I over an array A of size $8v\hat{b}$; let $I' \subseteq I$ denote the set of unsuccessful indices.

Step 3. Associate $3v^2$ processors with each cell of A and use these to $3v$-scatter the active elements in $\mathscr{B}_i \cap \bigcup_{j \in I \setminus I'} \mathscr{X}_j$ over A_i, for $i = 1, ..., m$.

The algorithm clearly runs in constant time on a TOLERANT PRAM using $O(n + v^3 b)$ processors and $O(n + v^3 b)$ memory cells. Since $|I| \leqslant b \leqslant \hat{b}$, the density of the multi-scattering in Step 2 is bounded by $1/2$ and, by definition, the densities of the multi-scatterings of well-supplied color classes in Step 3 are also bounded by $1/2$. Let us agree to call a cluster $\mathscr{X}_i$ with $i \in I'$ (i.e., i was unsuccessful in Step 2) *unsuccessful*. If we take D_1 for the index set of active elements in unsuccessful clusters and denote by D_2 the index

set of elements that are unsuccessful in Step 3, clearly $D = D_1 \cup D_2$.

For the proof of part (a), fix an arbitrary active element X_j in a well-supplied color class (i.e., $j \in B'$) and suppose that $X_j \in \mathscr{A}_i$. As argued above, $j \in D$ (i.e., X_j is unlucky) if and only if either $i \in I'$ (i.e., i is unsuccessful in Step 2) or $j \in D_2$ (i.e., X_j participates in Step 3, but is unsuccessful). But by Lemma 3.6(a), applied twice with $p \leqslant 1/2$, $\Pr(i \in I') \leqslant 2^{-4v}$ and $\Pr(j \in D_2) \leqslant 2^{-3v}$; hence $\Pr(j \in D) \leqslant 2^{-4v} + 2^{-3v} \leqslant 2^{-v}$.

For part (b), let R be a fixed nonempty subset of B', take $r = |R|$ and let $z \geqslant r/2^v$. Clearly $|R \cap D| \geqslant z$ only if either $|R \cap D_1| \geqslant z/2$ or $|R \cap D_2| \geqslant z/2$; we will consider these events separately and show each to be unlikely. First consider the multi-scattering in Step 2 and let $R' \subseteq I$ be the index set of clusters containing at least one active element whose index belongs to R; obviously $|R'| \leqslant r$. If $|R \cap D_1| \geqslant z/2$, i.e., at least $z/2$ elements of R are indices of elements in unsuccessful clusters, there must be at least $z/(2v)$ unsuccessful clusters containing elements with indices in R, i.e., $|R' \cap I'| \geqslant z/(2v)$. In other words, $\Pr(|R \cap D_1| \geqslant z/2) \leqslant \Pr(|R' \cap I'| \geqslant z/(2v))$. Since $z/(2v) \geqslant 2r \cdot 2^{-4v}$, we can apply Lemma 3.6(b) to bound the latter probability by $e^{-\zeta}$, where $\zeta = (z/(2v))^2/(32r \cdot 4v) \geqslant z^2/(2^9 r v^3)$. Since also $z/2 \geqslant 2r \cdot 2^{-3v}$, another application of Lemma 3.6(b) shows that $\Pr(|R \cap D_2| \geqslant z/2) \leqslant e^{-\zeta}$, where $\zeta = (z/2)^2/(32\,|R| \cdot 3v) \geqslant z^2/(2^9 rv)$. It follows that $|R \cap D| \geqslant z$ with probability at most $e^{-z^2/(2^9 r v^3)} + e^{-z^2/(2^9 rv)} \leqslant 2e^{-z^2/(2^9 r v^3)}$.

For part (c), note that S can be considered as a function of all the random choices made by the algorithm. Now, a change in a single random choice made in Step 2 affects at most 2 clusters. Each of the at most $2v$ elements in the affected clusters in turn affects at most $3v$ other elements in Step 3. Hence altogether at most $2v + 6v^2 \leqslant 8v^2$ of the output variables $y_1, \ldots, y_n$ are affected, and it is easy to see that no more output variables are affected by a change in a single random choice in Step 3. In other words, if D changes to D' in response to a change in a single random choice, we always have that $|D \triangle D'| \leqslant 8v^2$; by the Lipschitz condition imposed on S, this means that S changes by at most $8cv^2$. Since the algorithm makes at most $7vb \leqslant 7vn$ random choices altogether, an application of Corollary 2.3 yields that $S \leqslant \max\{2E(S), 4 \cdot 8cv^2 \cdot (7vn)^{5/8}\} = O(E(S) + cv^3 n^{5/8})$ with probability at least $1 - 2^{-n^{1/8}}$. ∎

In order to simplify the applications of Lemma 6.1 in the following, we discuss a generic application in detail at this point and introduce a convenient shorthand that will be used in later applications.

Assume that we are given a set $\mathscr{B}$ of active elements with colors in $\{1, \ldots, m\}$ and stored in an array Q of size n. For $i = 1, \ldots, m$, let $\mathscr{B}_i$ be the set of elements in $\mathscr{B}$ with color i. Further assume that we are given m disjoint arrays $A_1, \ldots, A_m$. Our goal is to place most of the elements of $\mathscr{B}_i$ in A_i, for $i = 1, \ldots, m$. To this end let x_j be the color of the element X_j in the jth cell of Q, for $j = 1, \ldots, n$, with $x_j = 0$ if the jth cell of Q does not contain any active element, and let v be a suitable positive integer (the choice of v will depend on our (application-specific) knowledge of the ratios $|A_i|/|\mathscr{B}_i|$). Then apply the algorithm of Lemma 6.1 to compute an incomplete placement $y_1, \ldots, y_n$ for $x_1, \ldots, x_n$ with bounds $|A_1|, \ldots, |A_m|$. Finally, for all $j \in \{1, \ldots, n\}$ with $x_j \neq 0$ and $y_j \neq 0$, actually place X_j in the y_jth cell of A_{x_j} (the algorithm of Lemma 6.1 already places X_j in this way; this is not specified in the statement of the lemma, however, so we repeat the operation here).

In what follows, an application of Lemma 6.1 as above will be called simply "v-compacting $\mathscr{B}_i$ to A_i, for $i = 1, \ldots, m$." Lemma 6.1 guarantees that a fixed element of a well-supplied color class will be unlucky with probability at most 2^{-v}. Furthermore, if the number of nonzero input numbers is $O(n/v^3)$, as in most applications of Lemma 6.1 in the present paper, then the algorithm uses $O(n)$ operations and $O(n)$ space.

In the remainder of the section we extend the definition of complete linear compaction to the case of several colors and prove a result corresponding to Theorem 4.6.

DEFINITION. For all $n, m \in \mathbb{N}$ and $d_1, \ldots, d_m \geqslant 0$, the *complete linear colored compaction* problem of size n and with limits $d_1, \ldots, d_m$ is, given n integers $x_1, \ldots, x_n$ in the range $0..m$ such that $|\{j: 1 \leqslant j \leqslant n \text{ and } x_j = i\}| \leqslant d_i$, for $i = 1, \ldots, m$, to compute a complete placement for $x_1, \ldots, x_n$ with bounds $O(d_1), \ldots, O(d_m)$.

The problem, discussed before the statement of Lemma 6.1, of allocating m disjoint arrays $A_1, \ldots, A_m$ is easy in a special case, namely when m is so small that the allocation can be done by means of brute-force prefix summation (Corollary 2.5). This leads to the following result.

THEOREM 6.2. *There is a constant $\varepsilon > 0$ such that for all given n, m, τ, $d_1, \ldots, d_m \in \mathbb{N}$ with $m = (\log n)^{O(1)}$ and $\tau \geqslant \log^* n$, complete linear colored compaction problems of size n and with limits $d_1, \ldots, d_m$ can be solved on a* TOLERANT PRAM *using $O(\tau)$ time, $\lceil n/\tau \rceil$ processors and $O(n + \sum_{i=1}^m d_i)$ space with probability at least $1 - 2^{-n^\varepsilon}$ (Las Vegas).*

Proof. We first devise a nonoptimal algorithm that solves the problem in $O(\log^* n)$ time using n processors and afterwards show that optimality can be achieved essentially as in the proof of Theorem 4.6. The basic idea is to place the active elements in $\log^* n$ successive stages, similarly as in the algorithm of Corollary 4.5. With a view towards a future application (Lemma 9.1), we first consider a more general setting, in which we ignore the difficulties of space allocation discussed above, but in return show how to tolerate $m = O(n^{1-\delta})$ different colors, for arbitrary fixed $\delta > 0$.

Without loss of generality assume that δ is rational and at most 1 and that $d_i \leqslant n$, for $i = 1, \ldots, n$ (otherwise the elements

in $\mathscr{B}_i$ can be deactivated in a trivial manner). Say that a color class $\mathscr{B}_i$ is *small* if $b_i \leqslant n^{\delta/2}$, and *large* otherwise, and note that the total number of elements in small color classes is $O(n^{1-\delta} \cdot n^{\delta/2}) = O(n^{1-\delta/2})$. We begin by reducing the number of active elements in large color classes to a similar level, but in a balanced way (i.e., each large color class loses most of its elements).

Define the *active fraction* as zero if all color classes are small, and otherwise as the maximum, over all large color classes $\mathscr{B}_i$, of the ratio of the number of (currently) active elements in $\mathscr{B}_i$ to b_i. We aim at decreasing the active fraction to at most $n^{-\delta/8}$, after which the total number of remaining active elements will be $O(n^{1-\delta/2} + n^{1-\delta/8}) = O(n^{1-\delta/8})$. We first show that if the active fraction has been reduced to at most v^{-3}, for some given $v \in \mathbb{N}$, then in constant time it can be reduced further to at most $\max\{2^{-3v}, n^{-\delta/8}\}$ with high probability. If $v \geqslant n^{\delta/24}$, there is nothing to show. Otherwise use the algorithm of Lemma 6.1 to $3v$-compact the remaining active elements in $\mathscr{B}_i$ to an array of size $\lceil 6 \cdot 3d_i/v^2 \rceil$, for $i = 1, \ldots, m$. Assuming that the active fraction is at most v^{-3}, Lemma 6.1 shows that the $3v$-compaction can be carried out in constant time using $O(n + v^3(n^{1-\delta/2} + \sum_{i=1}^{m}(b_i/v^3))) = O(n)$ processors and that for each fixed large color class $\mathscr{B}_i$ (which, by assumption, is well-supplied), the number of active elements in $\mathscr{B}_i$ after the $3v$-compaction is bounded by $\max\{b_i/2^{3v}, b_i^{3/4}\}$, except with probability at most $2e^{-\zeta}$, where $\zeta = (b_i^{3/4})^2/(2^9(b_i/v^3) \cdot (3v)^3) = \Omega(b_i^{1/2}) = \Omega(n^{\delta/4})$. With high probability, the fraction of active elements left in each large color class $\mathscr{B}_i$ after the $3v$-compaction therefore is at most $\max\{2^{-3v}, b_i^{-1/4}\} \leqslant \max\{2^{-3v}, n^{-\delta/8}\}$, for $i = 1, \ldots, m$, i.e., the active fraction has been reduced to the same level. Applying this procedure at most $\log^* n$ times with $v = 1, 2, 2^2, 2^{2^2}, \ldots$ with high probability reduces the active fraction to at most $n^{-\delta/8}$, as desired. Note that the successive arrays used by a color class can be taken as tightly packed subarrays of a single array.

In order to complete the compaction, we first use the algorithm of Corollary 4.3 to move the remaining $O(n^{1-\delta/8})$ active elements to an array of size $O(n^{1-\delta/8})$, after which constant time and n processors suffice to $\lfloor n^{\delta/8} \rfloor$-scatter the active elements of $\mathscr{B}_i$ over an array of size $2 \cdot \max\{d_i, \lceil n^{3\delta/4} \rceil\}$, for $i = 1, \ldots, m$. Since the number of active elements in a large color class $\mathscr{B}_i$ is at most $b_i/n^{\delta/8} \leqslant d_i/n^{\delta/8}$, the density of each of these multi-scatterings is bounded by $1/2$, so that Lemma 3.6(a) ensures that with high probability all elements are successful. At this point, for $i = 1, \ldots, m$, the elements of $\mathscr{B}_i$ are stored in an array of size $\lceil 18d_i/1^2 \rceil + \lceil 18d_i/2^2 \rceil + \lceil 18d_i/4^2 \rceil + \cdots + 2 \cdot \max\{d_i, \lceil n^{3\delta/4} \rceil\} = O(d_i + n^{3\delta/4})$. For all color classes $\mathscr{B}_i$ with $d_i \geqslant n^{3\delta/4}$, this compaction is sufficiently tight, while the remaining color classes can be compacted into linear space using the algorithm of Corollary 4.3 (recall that we have $\Theta(n^{\delta})$ processors for each color class).

Under the restriction $m = (\log n)^{O(1)}$ of the theorem, the allocation of an array to each color class, which was ignored above, can clearly be done in constant time using the algorithm of Corollary 2.5.

The algorithm described so far can be executed in $O(\log^* n)$ time with n processors. To achieve optimal speedup, it suffices, in light of Theorem 4.6, to show that the number of active elements can be reduced to $O(n/\tau)$ in $O(\tau)$ time. To this end assume that $\tau \leqslant (\log n)/32$ (otherwise sort the input numbers using the algorithm of Lemma 2.9), divide $\mathscr{X}$ into $\lceil n/\tau \rceil$ *clusters* of at most τ input elements each and associate a processor with each cluster. Then use the algorithm of Corollary 2.5 to allocate an array A_i of size $8d_i$ to $\mathscr{B}_i$, for $i = 1, \ldots, m$. Similarly as in the proof of Theorem 4.6, each processor now attempts in 2τ *rounds* to place the active elements of its cluster in the arrays corresponding to their colors. The argument in the proof of Theorem 4.6 shows that with high probability the number of active elements left after the last round is $O(n/\tau)$. ▮

7. INTERVAL ALLOCATION

While the compaction problem asks that unit intervals be placed in a base segment, the interval allocation problem, defined below, specifies intervals of varying length to be placed. Viewed another way, each input element is a request for a *block* of consecutive indices of a size given by the value of the request. Informally, condition (2) means that blocks do not overlap, and (3) means that the allocated blocks are optimally packed, except for a constant factor. Another difference between complete linear compaction and interval allocation is that in the case of compaction, an upper bound on the number of elements present (the limit d) is provided as part of the input, while in the case of interval allocation the choice of an appropriate size for the base segment is left to the algorithm. Since a suitable value for d could actually be computed using the algorithm of Theorem 5.3 as in the proof of Lemma 6.1, this difference is of no particular significance, but merely convenient for the exposition.

DEFINITION. For all $n \in \mathbb{N}$, the (complete) *interval allocation* problem of size n is the following: Given n nonnegative integers $x_1, \ldots, x_n$, compute n nonnegative integers $y_1, \ldots, y_n$ such that

(1) For $j = 1, \ldots, n$, $x_j = 0 \Leftrightarrow y_j = 0$;

(2) For $1 \leqslant i < j \leqslant n$, if $0 \notin \{x_i, x_j\}$, then $\{y_i, \ldots, y_i + x_i - 1\} \cap \{y_j, \ldots, y_j + x_j - 1\} = \varnothing$;

(3) $\max\{y_j : 1 \leqslant j \leqslant n\} = O(\sum_{j=1}^{n} x_j)$.

A natural extension of the interval allocation problem would augment the solution by an appropriate size for the base segment, i.e., by an integer s with $s \geqslant \max\{y_j + x_j - 1 : 1 \leqslant j \leqslant n\}$, but $s = O(\sum_{j=1}^{n} x_j)$. Such a quantity is usually

needed in applications of interval allocation, and our algorithm for interval allocation essentially generates it internally. By Lemma 2.6, however, a suitable choice for s (namely, $\max\{y_j + x_j - 1 : 1 \leqslant j \leqslant n\}$) can be computed from the output of interval allocation, as defined here, for which reason we have refrained from including it in the problem definition.

While interval allocation is a generalization of compaction, we will now show that interval allocation reduces to colored compaction. First note that if all nonzero requests (i.e., requests of nonzero value) are of value 1, i.e., if $x_j \in \{0, 1\}$, for $j = 1, ..., n$, then we can indeed first use the algorithm of Theorem 5.3 to compute a linear overestimate $\hat{b}$ for the number b of nonzero requests, and subsequently solve the complete linear compaction problem with input $x_1, ..., x_n$ and limit $\hat{b}$. Informally, what happens is that nonzero requests are interpreted as active elements in the usual sense; once the correct size of the destination array A has been established, a unit block (i.e., a single index) is associated with each cell of A, the active elements are placed in A by the compaction algorithm, and the block associated with a cell in A is allocated to the element, if any, placed in that cell (i.e., the requests are satisfied). The same approach works as long as all nonzero requests are of a common value l—we simply associate a block of l consecutive indices with each cell of A, rather than a single index.

If there are nonzero requests of m distinct values, we clearly have to use m distinct destination arrays $A_1, ..., A_m$, each associated with blocks of a different size; i.e., we have to resort to colored compaction. A number of difficulties have to be tackled in this generalization. First, we need to estimate the sizes of several color classes simultaneously; this can still be done by the algorithm of Theorem 5.3 if $m = O(n^{1-\delta})$, for some fixed $\delta > 0$. Second, the colored compaction can be carried out using the algorithm of Theorem 6.2, but only under the (more stringent) restriction $m = (\log n)^{O(1)}$. Third, the blocks of indices associated with $A_1, ..., A_m$ must be allocated from an appropriate base array. More precisely, with each array A_i we allocate a *segment* consisting of $|A_i|$ blocks of the appropriate size, i.e., of the size associated with the color i, after which the association of a single block with each cell in A_i is trivial. Since this allocation reduces to prefix summation, we aim to carry it out using the algorithm of Corollary 2.5. This requires, on the one hand, that $m = (\log n)^{O(1)}$, as above, and, on the other hand, that the sizes of the blocks to be allocated is polynomial in n. On the outset, these requirements are not satisfied: The numbers $x_1, ..., x_n$ could all be distinct, which would give us as many as n color classes, and they could be arbitrarily large. However, the input can be scaled and rounded to satisfy the requirements, as we show next.

First observe that if M denotes the maximum request value, i.e., $M = \max\{x_j : 1 \leqslant j \leqslant n\}$, then replacing each nonzero request value x_j by the nearest multiple of $u = \lceil M/n \rceil$ no smaller than x_j increases the sum W of all request values by at most $ub \leqslant M + b$, where b is the number of nonzero requests, i.e., at most triples W. As a result of this transformation, we can consider all request values to be integers in the range $0..n$ (simply measure requests and blocks in units of size u). If at this point we replace each nonzero (modified) request value by the nearest larger power of 2, W at most doubles, and only $O(\log n)$ different request values remain; i.e., the compaction-based algorithm sketched above becomes applicable with $m = O(\log n)$ color classes. Every modified request value is at least as large as the corresponding original value; since the modified values sum to at most six times the original sum, however, solving the interval allocation problem defined by the modified request values produces a solution to the original interval allocation problem.

The maximum request value M can be computed using the algorithm of Lemma 2.6. In fact, for the applications of Theorem 7.1 in the present paper, we will frequently have $M = O(n)$, in which case it is not necessary to actually compute M (because the procedure described above, with trivial modifications, can be employed with $u = 1$ whenever M is polynomial in n). Summing up, we have seen that interval allocation reduces to complete linear colored compaction with a logarithmic number of colors.

THEOREM 7.1. *There is a constant $\varepsilon > 0$ such that for all given n, $\tau \in \mathbb{N}$ with $\tau \geqslant \log^* n$, interval allocation problems of size n can be solved on a* TOLERANT PRAM *using $O(\tau)$ time, $\lceil n/\tau \rceil$ processors and $O(n)$ space with probability at least $1 - 2^{-n^{\varepsilon}}$ (Monte Carlo).*

Proof. By the discussion above and Theorem 6.2. In particular, note that since resources (indices) are divided optimally between color classes and with a constant-factor "waste" within color classes, the overall "waste factor" is bounded by a constant, as required by condition (3) in the definition of interval allocation. ∎

Remark. Part of the development of Theorem 7.1 took place in a dialog with Joseph Gil. An earlier version of the present paper achieved running times of $O(\log \log n \log^* n / \log \log \log n)$, the bottleneck being compaction. After receiving a preliminary sketch of the algorithm of Theorem 7.1 geared towards this running time, Gil informed us of the results of Matias and Vishkin (1991), unpublished at the time, and observed their applicability in the context of the algorithm, which allowed him to derive a first interval allocation algorithm with a running time of $O(\log^* n)$. We improved his result by giving an algorithm with optimal speedup and a lower failure probability and by implementing the algorithm on the weaker TOLERANT PRAM, after which the communication with Gil ceased. Theorem 7.1 states the last result mentioned above, except for a still smaller failure probability. A slightly weaker result

was published in (Gil *et al.*, 1991). A time bound of $O(\log \log n)$ for a less general *load balancing* problem was shown by Gil (1990, 1994).

Remark. At this point we can give only a Monte Carlo algorithm for interval allocation, and Theorem 7.1 is formulated accordingly. This is due to the use of the Monte Carlo fine-profiling algorithm of Theorem 5.3. Using Corollary 10.5 instead of Theorem 5.3, however, allows us to obtain a Las Vegas algorithm for interval allocation. The same remark applies to Theorem 7.2 below.

Whereas the use of Theorem 7.1 in memory allocation is obvious, one additional observation is needed for its application to the allocation of processors. The reason is that a processor is an active device that needs to know about the task that it is to execute. Theorem 7.1 can be used to communicate this information to the first processor in each *team*, i.e., in each group of consecutively numbered processors allocated to a common task, but the information must subsequently be broadcast to the remaining processors in each team. In recognition of this fact, we consider a slight variation of the interval allocation problem called the *interval marking* problem. In the definition below, informally, $x_1, ..., x_n$ are the sizes of n requests for processors. The output consists of a size indicator s together with s integers $z_1, ..., z_s$ and specifies the allocation of s virtual processors $P_1, ..., P_s$ as follows: For $j = 1, ..., s$, the meaning of $z_j = i \in \{1, ..., n\}$ is that P_j is a member of the team allocated to the ith request; the meaning of $z_j = 0$ is that P_j is not part of any team. Condition (1) requires the processors in each team to be consecutively numbered, (2) expresses that the number of virtual processors in the team allocated to the ith request is indeed exactly x_i, for $i = 1, ..., n$, and (3) states that the total number of virtual processors exceeds the number of requested processors by at most a constant factor; this allows the allocated processors to be simulated without loss, up to a constant factor, by any number of available physical processors.

DEFINITION. For all $n \in \mathbb{N}$, the *interval marking* problem of size n is the following: Given n nonnegative integers $x_1, ..., x_n$, compute nonnegative integers s, $z_1, ..., z_s$ such that

(1) For all integers i, j, k with $1 \leqslant i \leqslant j \leqslant k \leqslant s$, if $z_i = z_k \neq 0$, then $z_j = z_i$;

(2) For $i = 1, ..., n$, $|\{j: 1 \leqslant j \leqslant s \text{ and } z_j = i\}| = x_i$;

(3) $s = O(\sum_{j=1}^{n} x_j)$.

In applications of interval marking, it is essential for each processor to know its relative position within its team. This information does not appear explicitly in the problem definition. Since the processors in each team are consecutively numbered, however, each processor in a team can compute its relative position as the difference between its own number and the smallest number of a processor in the team; the latter quantity can be made available in a cell indexed by the number of the request to which the team is allocated.

In most applications of interval marking in the present paper we will have $\sum_{j=1}^{n} x_j = O(n)$. We next show that under this restriction, we can solve the interval marking problem with input $x_1, ..., x_n$ in constant deterministic time with n processors after solving the interval allocation problem with input $x_1, ..., x_n$ using the algorithm of Theorem 7.1. The reader may think of this as a reduction of interval marking to interval allocation; this is not quite exact, however, since we will use a special property of the solution produced by Theorem 7.1 (conversely, under the restriction $\sum_{j=1}^{n} x_j = O(n)$, it is easy to show that interval allocation reduces to interval marking).

We view interval allocation with input $x_1, ..., x_n$ as allocating disjoint subarrays $A_1, ..., A_n$ of sizes $x_1, ..., x_n$ from a base array and note that it is trivial to mark the first cell of A_i with the integer i, for $i = 1, ..., n$. As already discussed above, the corresponding interval marking problem can essentially be solved by copying the integer stored in the first cell of A_i to the remaining cells of A_i, for $i = 1, ..., n$. In other words, it suffices to provide each cell of a subarray with a pointer to the beginning of the subarray. Now recall that our algorithm for interval allocation actually allocates all subarrays from $O(\log n)$ segments, each of which consists of tightly packed subarrays of the same size. If we store the subarray size of each segment in the first cell of the segment, which is easy to do, it suffices to provide each cell of a segment with a pointer to the beginning of the segment, since with this pointer and the relevant subarray size it can easily compute the beginning of its subarray. We are now left with a problem that can be viewed as an instance of the segmented broadcasting problem defined in Section 2; each beginning of a segment corresponds to one nonzero input bit. If $\sum_{j=1}^{n} x_j = O(n/h)$, where $h = \lfloor n^{1/4} \rfloor$, the segmented broadcasting problem is of size $O(n/h)$, and we can solve it using negligible resources, according to part (b) of Lemma 2.8. In the opposite case, on the other hand, we can ensure that the length of each of the $O(\log n)$ segments mentioned above is a multiple of h by appending a suitable number of dummy cells—this increases the total length by $O(h \cdot \log n) = O(n/h) = O(\sum_{j=1}^{n} x_j)$, i.e., by at most a constant factor. Then, however, the output of the segmented broadcasting problem will be constant within blocks of size h; i.e., we have in effect reduced the size of the segmented broadcasting problem by a factor of h, and we can again appeal to part (b) of Lemma 2.8 and obtain the following result.

THEOREM 7.2. *There is a constant $\varepsilon > 0$ such that for all given n, $\tau \in \mathbb{N}$ with $\tau \geqslant \log^* n$, interval marking problems of size n can be solved on a* TOLERANT PRAM *using $O(\tau)$*

time, $\lceil (n+W)/\tau \rceil$ processors and $O(n+W)$ space with probability at least $1-2^{-n^c}$ (Monte Carlo), where W is the sum of the input numbers.

Proof. By the discussion above, Lemma 2.8(b) and Theorem 7.1. The dependence of the resource bounds on W is due to the fact that the size of the output is $O(W+1)$. ∎

The proof of Theorem 7.2 shows how to allocate processors to requesting tasks, each of which requests one or more processors. A situation frequently encountered is that many tasks are so small that they do not require "an entire processor," while at the same time the number of tasks is so large that we cannot afford to allocate a processor to each. Theorem 7.2 easily extends to cover this situation as well. Let $n, \tau \in \mathbb{N}$ with $\tau \geqslant \log^* n$, assume that we are given a collection $\mathscr{T}_1, \ldots, \mathscr{T}_n$ of n tasks, and suppose that for $j=1, \ldots, n$, we know a positive integer q_j, called the *length* of $\mathscr{T}_j$, such that $\mathscr{T}_j$ can be executed in $O(q_j)$ time with one processor, or in $O(\tau)$ time with $\lceil q_j/\tau \rceil$ processors. Then the n tasks can be executed in $O(\tau)$ time with $\lceil W/\tau \rceil$ processors, where $W=\sum_{j=1}^n q_j \geqslant n$. To see this, define $\mathscr{T}_j$ to be *small* if $q_j \leqslant \tau$, and *large* otherwise, for $j=1, \ldots, n$. Begin by allocating $\lfloor q_j/\tau \rfloor$ processors to $\mathscr{T}_j$, for $j=1, \ldots, n$, clearly a total of at most W/τ processors. For $j=1, \ldots, n$, if $\mathscr{T}_j$ is large, then $\lfloor q_j/\tau \rfloor \geqslant \frac{1}{2}\lceil q_j/\tau \rceil$, so that the processors allocated to $\mathscr{T}_j$ suffice to execute $\mathscr{T}_j$ in $O(\tau)$ time. What remains is to execute the small tasks. Partition these into $m=\lceil n/\tau \rceil$ *groups* $G_1, \ldots, G_m$ of at most τ tasks each. For $i=1, \ldots, m$, compute the *total length* Q_i of G_i as the sum of the lengths of the tasks in G_i. Then allocate $\lceil Q_i/\tau \rceil$ processors to G_i, for $i=1, \ldots, m$, a total of at most $m+W/\tau \leqslant 2\lceil W/\tau \rceil$ processors. It is not difficult to see that using sequential prefix summation, the tasks in each group can be distributed among the processors allocated to the group in such a way that each processor receives tasks of total length $O(\tau)$. All that remains is to let each processor execute the tasks given to it sequentially in $O(\tau)$ time.

When invoking the principle above, we will speak of "operation allocation" rather than processor allocation.

While Theorems 7.1 and 7.2 are our main results concerning the interval allocation and interval marking problems, we also need a more technical lemma (Lemma 7.3 below) that parallels Lemma 6.1 and allows us to perform what we call *incomplete allocation* in constant time. Just as Lemma 6.1 claims efficient deactivation only of elements in well-supplied color classes, those for which the available array is at least $6v$ times larger than the number of elements to be placed there, Lemma 7.3 is wasteful in a sense that we make explicit through the introduction of a so-called *slack parameter*. Incomplete allocation, recognized independently by Gil *et al.* (1991) as a useful technique, was introduced and first used in the hashing algorithm of (Bast and Hagerup, 1991).

DEFINITION. For all $n \in \mathbb{N}$ and $\lambda \geqslant 1$, an *incomplete interval placement* with *slack* λ for n nonnegative integers $x_1, \ldots, x_n$ is a sequence $y_1, \ldots, y_n$ of n nonnegative integers such that

(1) For $j=1, \ldots, n$, if $x_j=0$, then $y_j=0$;

(2) For $1 \leqslant i < j \leqslant n$, if $0 \notin \{y_i, y_j\}$, then $\{y_i, \ldots, y_i+x_i-1\} \cap \{y_j, \ldots, y_j+x_j-1\} = \varnothing$;

(3) $\max\{y_j: 1 \leqslant j \leqslant n\} = O(\lambda \cdot \sum_{j=1}^n x_j)$.

The set $\{j: 1 \leqslant j \leqslant n, x_j \neq 0 \text{ and } y_j = 0\}$ is called the *residue set* of the incomplete interval placement. If the residue set is empty, the interval placement is *complete*.

Contrasted with the definition of (complete) interval allocation, the definition above does not require a block to be allocated to every request, and blocks may be allocated from a range λ times as large. An algorithm that computes complete interval placements with constant slack performs standard interval allocation.

LEMMA 7.3. *There is a constant $\varepsilon > 0$ such that for all given $n, v, \tau \in \mathbb{N}$, an incomplete interval placement for n given nonnegative integers $x_1, \ldots, x_n$ with slack v can be computed on a* TOLERANT PRAM *using $O(\tau)$ time, $\lceil (n+v^3W)/\tau \rceil$ processors and $O(n+v^3W)$ space with probability at least $1-2^{-n^\varepsilon}$ (Monte Carlo), where $W=\sum_{j=1}^n x_j$, such that the residue set D of the placement satisfies the following*:

(a) *For $j=1, \ldots, n$, $\Pr(j \in D) \leqslant 2^{-v}$;*

(b) *For every fixed nonempty subset R of $\{1, \ldots, n\}$ and for all $z \geqslant |R|/2^v$,*

$$\Pr(|R \cap D| \geqslant z) \leqslant 2e^{-z^2/(2^9|R|v^3)};$$

(c) *For every nonnegative real function S of D that satisfies a Lipschitz condition with constant c,*

$$S = O(E(S) + cv^3 n^{5/8})$$

with probability at least $1-2^{-n^{1/8}}$.

Proof. The reduction of interval allocation to complete linear colored compaction with a logarithmic number of colors extends to incomplete interval allocation and incomplete colored compaction in a straightforward way. Once each request has been marked with its color, an integer in the range $1..m$, we can use the algorithm of Theorem 5.3 to compute a linear overestimate $\hat{b}_i$ for the size of $\mathscr{B}_i$, for $i=1, \ldots, m$, and that of Corollary 2.5 to allocate arrays $A_1, \ldots, A_m$ with $|A_i| = 6v\hat{b}_i$, for $i=1, \ldots, m$, and their associated blocks. Since $\sum_{i=1}^m \hat{b}_i = O(W)$, both the arrays and the blocks can be allocated from a base array of size $O(vW)$. Lemma 7.3 now follows easily from Lemma 6.1; in particular, note that since the number of nonzero input numbers is bounded by W, the number of operations and memory cells needed is $O(n+v^3W)$. ∎

Because of the near-equivalence of interval allocation and interval marking, Lemma 7.3 can be used for incomplete allocation of processors as well as of memory cells. Extending our earlier terminology, we call an input element *unlucky* in an application of Lemma 7.3 if its index belongs to the residue set, and *lucky* otherwise. Just as we introduced the concept of v-compaction to facilitate the application of Lemma 6.1, let us agree to use the term "v-allocation," for $v \in \mathbb{N}$, to denote an application of Lemma 7.3 with slack v, followed by the actual allocation of memory cells or processors to the lucky elements. By Lemma 7.3(a), the probability that a fixed element is unlucky in a v-allocation is at most 2^{-v}, and if the "total resource demand" $W = \sum_{j=1}^{n} x_j$ is $O(n/v^3)$, then the v-allocation uses $O(n)$ operations and $O(n)$ space.

8. COARSE-PROFILING

While many applications call for the profiling of sequences of values in the range $1..n$ stored in an array of size n, our best strong fine-profiling result (Theorem 5.3) allows only $O(n^{1-\delta})$ different values, for fixed $\delta > 0$. It is hence necessary to relax the requirements imposed on a profile. Whereas the definition of fine-profiling seems quite natural, it is not obvious how to define a computationally more tractable profiling problem. The following definition of a *coarse-profile*, which at first glance may seem rather artificial, turns out to be useful.

DEFINITION. Let $n, m \in \mathbb{N}$, and let $x_1, ..., x_n$ be n integers in the range $0..m$. For $i = 1, ..., m$, take $b_i = |\{j: 1 \leqslant j \leqslant n \text{ and } x_j = i\}|$. An *$m$-color coarse-profile* for $x_1, ..., x_n$ is a sequence of m independent nonnegative integer random variables $\hat{b}_1, ..., \hat{b}_m$ such that

(A) $\sum_{i=1}^{m} \hat{b}_i = O(n)$;

(B) For $i = 1, ..., m$ and for all $a \geqslant 1$, $\Pr(b_i > a\hat{b}_i) \leqslant 2^{-a}$.

For $n, m \in \mathbb{N}$, the m-color coarse-profiling problem of size n is to compute an m-color coarse-profile for n given integers in the range $0..m$.

We will refer to condition (A) in the definition above as the *linear-sum condition*. As in the case of fine-profiling, input elements of value 0 are "dummy elements" that do not take part in the profiling.

In the following, we show that n-color coarse-profiling problems of size n can be solved with optimal speedup in $O(\log^* n)$ time. We first explain the main ideas in the context of an algorithm that uses n processors and later indicate how to achieve optimal speedup. We begin by tackling a simpler problem, that of estimating just the large multiplicities. Our approach is to extrapolate from a fine-profile for a random sample of size $n^{1-\gamma}$, for some suitably chosen constant $\gamma > 0$. One small complication is that although the sample certainly contains at most $n^{1-\gamma}$ distinct values, these are spread out over the entire range of size n, whereas for an application of Theorem 5.3 a much smaller range is required. The following technical lemma, which will be used frequently in the remainder of the paper, allows us to rename the sample values as required, i.e., to compute an injective function (namely, $i \mapsto y_i$) from the set of original sample values to a range of size $O(n^{1-\gamma})$.

LEMMA 8.1. *For all given $n, d, s, \tau \in \mathbb{N}$ with $d = O(n/\tau)$, the following problem reduces, using $O(\tau)$ time, $\lceil n/\tau \rceil$ processors and $O(n)$ space on a* TOLERANT PRAM, *to a complete compaction problem of size n and with parameters $d \to_s 0$: Given d integers $x_1, ..., x_d$ in the range $0..n$, compute n nonnegative integers $y_1, ..., y_n$ such that*

(1) *For $i = 1, ..., n$, $y_i \neq 0 \Leftrightarrow i \in \{x_j: 1 \leqslant j \leqslant d\}$;*

(2) *For $1 \leqslant i < j \leqslant n$, if $y_i \neq 0$, then $y_i \neq y_j$;*

(3) $\max\{y_i: 1 \leqslant i \leqslant n\} \leqslant s$.

Proof. Just as we associate a record X_j with value x_j with each input variable x_j, let us associate an *output record* Y_i with value y_i with the output variable y_i, for $i = 1, ..., n$.

The problem is simply to mark those output records that are to receive nonzero values, since afterwards the problem can be solved by compacting the marked records. Whereas the marking would be trivial on the ARBITRARY PRAM, on the TOLERANT PRAM several occurrences of a value i might prevent the marking of Y_i. Our solution is to use guarded writing of a new kind that we call *inverted guarded writing*. For $\tau > 1$ we do not have a (physical) guard processor for each output record, which is the reason why guarded writing of the kind employed in previous sections cannot be used. Instead note that by assumption, we can associate a processor with each element $j \in \{1, ..., d\}$. If $x_j \neq 0$, then let this processor continuously write some value to Y_{x_j}. At the same time associate a virtual guard processor with each output record and let $\lceil n/\tau \rceil$ physical processors simulate the virtual guard processors in $O(\tau)$ time. If each virtual guard processor attempts to modify the value stored in its associated output record and marks the record if and only if this fails, the desired marking will result. ∎

In the theorem below, condition (1) says that every nonzero estimate is a linear overestimate for the multiplicity that it estimates. Condition (2) ensures that nonzero estimates are in fact obtained at least for the large multiplicities.

THEOREM 8.2. *For every fixed $\delta > 0$ there is a constant $\varepsilon > 0$ such that the following holds: Let $n, \tau \in \mathbb{N}$ be given, let $x_1, ..., x_n$ be n given integers in the range $0..n$ and take $b_i = |\{j: 1 \leqslant j \leqslant n \text{ and } x_j = i\}|$, for $i = 1, ..., n$. Then it is possible, with probability at least $1 - 2^{-n^{\varepsilon}}$ (Monte Carlo) and on a* TOLERANT PRAM *using $O(\tau)$ time, $\lceil n/\tau \rceil$ processors and $O(n)$ space, to compute n independent nonnegative integer random variables $\hat{b}_1, ..., \hat{b}_n$ such that conditions (1)*

and (2) *below hold for some constant* $K \geq 1$ *and for each* $i \in \{1, ..., n\}$.

(1) $\hat{b}_i > 0 \Rightarrow b_i \leq \hat{b}_i \leq Kb_i$;

(2) $b_i \geq n^{\delta} \Rightarrow \hat{b}_i > 0$.

Proof. Without loss of generality assume that δ is rational and at most 1. By Lemma 2.9, we can also assume that $\tau \leq n^{\delta/4}$, so that at least $n^{1-\delta/4}$ processors are available. Let $h = \lceil n^{\delta/4} \rceil$ and carry out the following algorithm:

Step 1. Draw a random sample $\mathcal{Y}$ of $\mathcal{X}$ by including each input element in $\mathcal{Y}$ independently of other elements and with probability $1/h$. For $i = 1, ..., n$, let $b_i^{\mathcal{Y}} = |\mathcal{B}_i \cap \mathcal{Y}|$.

Step 2. Use the algorithm of Theorem 5.3 to estimate $b_i^{\mathcal{Y}}$, for $i = 1, ..., n$. First place the elements of $\mathcal{Y}$ in an array of size $O(n^{1-\delta/4})$. Since $|\mathcal{Y}| = O(n^{1-\delta/4})$ with high probability by Chernoff bound (a), this can be done by the algorithm of Corollary 4.3. Then use the algorithms of Lemma 8.1 and Corollary 4.3 to replace the values represented among the elements in $\mathcal{Y}$ by values in a range of size $O(n^{1-\delta/4})$. The algorithm of Theorem 5.3 now provides estimates $\hat{b}_1^{\mathcal{Y}}, ..., \hat{b}_n^{\mathcal{Y}}$ such that with high probability, $b_i^{\mathcal{Y}} \leq \hat{b}_i^{\mathcal{Y}} \leq K' b_i^{\mathcal{Y}}$, for $i = 1, ..., n$ and for some constant K' (take $\hat{b}_i^{\mathcal{Y}} = 0$ for all $i \in \{1, ..., n\}$ with $b_i^{\mathcal{Y}} = 0$).

Step 3. For $i = 1, ..., n$, if $\hat{b}_i^{\mathcal{Y}} \geq n^{\delta/2}$, then take $\hat{b}_i = 2h\hat{b}_i^{\mathcal{Y}}$; otherwise take $\hat{b}_i = 0$.

In order to analyze the algorithm, fix $i \in \{1, ..., n\}$. If $b_i \geq n^{\delta/2}$, then by Lemma 2.1, with high probability $b_i/(2h) \leq b_i^{\mathcal{Y}} \leq 2b_i/h$ and hence $b_i/(2h) \leq \hat{b}_i^{\mathcal{Y}} \leq 2K' b_i/h$, from which it follows that either $\hat{b}_i = 0$ or $b_i \leq \hat{b}_i \leq 4K' b_i$. If $b_i \geq n^{\delta}$, clearly with high probability $\hat{b}_i > 0$. On the other hand, if $b_i < n^{\delta/2}$, then with high probability $b_i^{\mathcal{Y}} < n^{\delta/2}/K'$, hence $\hat{b}_i^{\mathcal{Y}} < n^{\delta/2}$ and $\hat{b}_i = 0$. ∎

LEMMA 8.3. *There is a constant* $\varepsilon > 0$ *such that for all given* $n \in \mathbb{N}$, *n-color coarse-profiling problems of size n can be solved on a* TOLERANT PRAM *using* $O(\log^* n)$ *time, n processors and* $O(n)$ *space with probability at least* $1 - 2^{-n^{\varepsilon}}$ *(Monte Carlo).*

We begin by describing the algorithm informally. As in the profiling algorithm of Theorem 5.3, the basic idea is that a GCS for each color can be used to estimate the multiplicity of that color. Assume that we estimate the size of each color class by determining a 1-row of a GCS of the color class with parameters $r \times s$. Then r should be $\Theta(\log n)$, since color classes may be of size up to n. As can be seen from Lemma 3.3, increasing s has the effect of increasing the reliability of estimates derived from a 1-row of the GCS. It turns out, however, that choosing s as a constant already yields sufficiently reliable estimates to provide a coarse-profile.

The outstanding problem is how to determine a 1-row for each of the n graduated conditional scatterings. A complete evaluation of each GCS, as per Lemma 3.4, would use $\Theta(n \log n)$ processors and $\Theta(n \log n)$ space, whereas we want to get by with a linear amount of resources. Observe, however, that a small color class is likely to have a small 1-row, which can be found using fewer than $\Theta(\log n)$ resources by evaluating only the first few rows of its GCS. At the outset we do not know which of the color classes are small (indeed, we are in the process of estimating their sizes). But consider the GCS of a particular color class U and assume that it fills some row, say of index v. The GCS then has a 1-row no smaller than v, and thus Lemma 3.3(a) implies that $|U|$ is unlikely to be much smaller than 2^v, so that spending $\Theta(2^v)$ resources on the GCS of U is okay. If the GCS does not fill its row 2^v, it has a 1-row between v and 2^v, and such a 1-row can be determined in constant time by the algorithm of Lemma 3.4(a) using the $\Theta(2^v)$ resources justified above. If the GCS fills its row 2^v as well, on the other hand, this indicates that still more resources can be devoted to U. Each color class thus inspects "sample" rows $1, 2^1, 2^{2^1}, ...$ of its GCS until encountering a row that is not full; this takes $O(\log^* n)$ time. Processors and space are then allocated to the color classes accordingly, and the 1-rows of all color classes are determined in constant time.

We now provide a more formal description of the algorithm and begin by using the algorithm of Theorem 8.2 to obtain accurate estimates for all color classes of sizes $n^{1/4}$ or more, whose elements can subsequently be replaced by "dummy elements" of value 0 (see the discussion preceding the definition of a fine-profile in Section 5). We can therefore assume that $b_i < n^{1/4}$, for $i = 1, ..., n$. Associate a processor with each element of nonzero value, and then execute the following steps, where $r = \lceil (\log n)/4 \rceil$ and $K = 60$:

```
(1)  for i ∈ {1, ..., n} pardo
(2)    begin
(3)      Let the elements of B_i carry out a GCS S_i
(4)        with parameters r × 6;
(5)      (u_i, v_i) := (0, 1);
(6)      while v_i < r and row v_i of S_i is full do
(7)        (u_i, v_i) := (v_i, min{2^{v_i}, r});
(8)      Allocate 6v_i processors and 6v_i memory cells to B_i;
(9)      l_i := a 1-row of S_i with u_i ≤ l_i ≤ v_i;
(10)     b̂_i := K · 2^{l_i};
(11)   end;
```

The statements in lines (5) and (7) are simultaneous assignments to u_i and v_i; it is easy to see that after the first iteration of the while loop, u_i will always hold the value of v_i in the previous iteration. Since the graduated conditional scatterings carried out for different color classes are independent, the random variables $\hat{b}_1, ..., \hat{b}_n$ are clearly also independent, as required. Note also that when line (9) is executed, there is indeed a 1-row of $\mathcal{S}_i$ between u_i and v_i; this

follows easily from the observation that if $u_i > 0$, then row u_i is full, while if $v_i < r$, then row v_i is not full.

The allocation of processors and space in line (8) of the algorithm can be done in $O(\log^* n)$ time using the algorithms of Theorems 7.1 and 7.2, and these resources enables the computation of l_i in line (9) in constant time using the algorithm of Lemma 3.4(a). The algorithm therefore runs in $O(\log^* n)$ time. The lemmas below show that the resources allocated are not excessive (Lemma 8.5) and that the sequence $\hat{b}_1, ..., \hat{b}_n$ is indeed a coarse-profile (Lemmas 8.4 and 8.6).

LEMMA 8.4. *With high probability,* $\sum_{i=1}^{n} \hat{b}_i = O(n)$.

Proof. Fix $i \in \{1, ..., n\}$. Since always $\hat{b}_i \leqslant K \cdot 2^r = O(n^{1/4})$, by a martingale argument it suffices to show that $E(\hat{b}_i) = O(b_i + 1)$. If $b_i = 0$, all rows of $\mathscr{S}_i$ are nonfull and 0 is the only 1-row of $\mathscr{S}_i$, so that $\hat{b}_i = K$. Otherwise we find that

$$\begin{aligned} E(\hat{b}_i) &= \sum_{j=0}^{\infty} \Pr(\hat{b}_i > j) \\ &= \sum_{j=0}^{Kb_i - 1} \Pr(\hat{b}_i > j) + \sum_{j=Kb_i}^{\infty} \Pr(\hat{b}_i > j) \\ &\leqslant Kb_i + \sum_{l=0}^{\infty} 2^l Kb_i \cdot \Pr(\hat{b}_i > 2^l Kb_i). \end{aligned}$$

Now by Lemma 3.3(a), $\Pr(\hat{b}_i > 2^l Kb_i) = \Pr(6 \cdot 2^{l_i} > 6 \cdot 2^l b_i \geqslant 6) \leqslant (2e/(6 \cdot 2^l))^6 \leqslant 2^{-6l}$, for all $l \geqslant 0$, and it follows that

$$E(\hat{b}_i) \leqslant Kb_i + \sum_{l=0}^{\infty} 2^l Kb_i \cdot 2^{-6l} = O(b_i). \quad \blacksquare$$

LEMMA 8.5. *With high probability, the algorithm uses* $O(n)$ *processors and* $O(n)$ *space.*

Proof. For $i = 1, ..., n$, $v_i \leqslant 2^{u_i} \leqslant 2^{l_i}$. Therefore the total amount of processors and space used by the algorithm is $O(n + \sum_{i=1}^{n} v_i) = O(n + \sum_{i=1}^{n} 2^{l_i}) = O(n + \sum_{i=1}^{n} \hat{b}_i) = O(n)$ with high probability, where the last relation follows from the previous lemma. $\blacksquare$

LEMMA 8.6. *For* $i = 1, ..., n$ *and for all* $a \geqslant 1$, $Pr(b_i > a\hat{b}_i) \leqslant 2^{-a}$.

Proof. $b_i > a\hat{b}_i$ is equivalent to $b_i > aK \cdot 2^{l_i}$, which implies that $l_i < \log b_i \leqslant \log(n^{1/4}) \leqslant r$. Hence by Lemma 3.3(b),

$$\begin{aligned} \Pr(b_i > a\hat{b}_i) &= \Pr(l_i < r \text{ and } b_i > (aK/6) \cdot 6 \cdot 2^{l_i}) \\ &\leqslant 6 \cdot 2^{1 - Ka/12} = 12 \cdot 2^{-5a} \leqslant 2^{-a}. \end{aligned}$$

This ends the proofs of Lemmas 8.6 and 8.3. $\blacksquare$

We now describe a simple procedure called *scattering in time* that will be used on four separate occasions. Since in each case we shall need different properties of the procedure, we believe that it serves little purpose to list at this point all the properties of the procedure that we shall ever need. After describing the procedure, we therefore analyze it only with respect to its resource requirements; later we will refer back to the procedure and derive whatever properties are of interest. In three of the four cases, scattering in time is used as a "profile enhancer"; i.e., informally, it inputs a profile $\hat{b}_1, ..., \hat{b}_n$ and produces a "better" profile $\tilde{b}_1, ..., \tilde{b}_n$. In the fourth case we input a very good profile and use it to semisort.

Scattering in time takes as input n input elements $X_1, ..., X_n$ (the *primary input*) with values in the range $0..n$ (input elements with a value of 0 are dummy elements signifying "no element") and n nonnegative integers $\hat{b}_1, ..., \hat{b}_n$ (the *profile input*) with $\sum_{i=1}^{n} \hat{b}_i = O(n)$, as well as an integer τ (the *phase count*) with $\log^* n \leqslant \tau \leqslant \sqrt{n}$. As usual, let $\mathscr{B}_i$ be the set of input elements of value i, for $i = 1, ..., n$. We begin by using the algorithm of Theorem 7.1 to allocate an array A_i of size $\hat{b}_i$ to $\mathscr{B}_i$, for $i = 1, ..., n$, each cell of which contains a counter, initialized to zero, and a list header, initially denoting an empty list. For $i = 1, ..., n$, every element of $\mathscr{B}_i$ now chooses a random integer, called its *list number*, from the set $\{1, ..., \hat{b}_i\}$, and another random integer, called its *phase number*, from the set $\{1, ..., \tau\}$. By Chernoff bound (a), with high probability the set $\mathscr{Y}_l$ of elements with phase number l is of size $O(n/\tau)$, for $l = 1, ..., \tau$. Using the algorithm of Theorem 6.2 if $\tau \leqslant \log n$ and that of Lemma 2.9 if $\tau > \log n$, we can therefore store the elements of $\mathscr{Y}_l$ in an array Q_l of size $O(n/\tau)$, for $l = 1, ..., \tau$.

The arrays $Q_1, ..., Q_\tau$ are next processed one by one. To process an array Q_l means to associate a processor with each cell in Q_l, and then to process all cells in Q_l simultaneously and in constant time. If a cell of Q_l is empty or contains a dummy element, the processor in charge of that cell does nothing. Otherwise, suppose that the element X stored in the cell belongs to $\mathscr{B}_i$ and chose j as its list number. The processor then attempts to increment the counter stored in $A_i[j]$ by 1. If this fails, i.e., if some other processor attempts to increment the same counter in the same time step, X is said to *collide*. Otherwise X is *noncolliding*, and the processor in charge of X proceeds to insert X in the list whose header is stored in $A_i[j]$. An important fact to note is that an element X of $\mathscr{B}_i$ collides exactly if some other element of $\mathscr{B}_i$ chooses both the same list number and the same phase number as X, for $i = 1, ..., n$, i.e., the scattering in time of $\mathscr{B}_i$ can be analyzed as a 1-scattering of $\mathscr{B}_i$ over an array of $\tau\hat{b}_i$ cells; the advantage of scattering in time is that it uses less space and (therefore) fewer operations.

Once a first pass as described above has been completed, we shall sometimes carry out a second pass exactly like the first pass, except that the only elements taking part in the

computation are those that collided in the first pass. In either case, we say that an element is *successful* if and only if there is some pass in which it does not collide. It is easy to see that one-pass or two-pass scattering in time can be carried out with high probability using $O(\tau)$ time, $\lceil n/\tau \rceil$ processors and $O(n)$ space. After the scattering in time, the set $\mathscr{X}'$ of successful elements as well as all counters and lists are available for further processing. In particular, note that for all $i \in \{1, ..., n\}$ such that $\hat{b}_i = \tau^{O(1)}$, we can compute $|\mathscr{B}_i \cap \mathscr{X}'|$ exactly in $O(\tau)$ time using operation allocation, as described in Section 7. It suffices to observe that if $\hat{b}_i = \tau^{O(1)}$, Lemma 2.4 can be used to compute the sum of the counters stored in A_i, either in $O(\hat{b}_i)$ time with one processor, or in $O(\tau)$ time with $\lceil \hat{b}_i/\tau \rceil$ processors. Since $\sum_{i=1}^n \hat{b}_i = O(n)$, the resource requirements remain as stated above.

Our first application of scattering in time is to the computation of a profile with the somewhat unnatural properties listed in Lemma 8.7 below. Once Lemma 8.7 has been established, a second application of scattering in time will allow us to obtain a coarse-profiling algorithm with optimal speedup. The proof of Lemma 8.7 is rather technical, but the main ideas behind it are as follows: We already have an n-processor coarse-profiling algorithm (Lemma 8.3). In the context of an algorithm with optimal speedup and a running time of $\Theta(\tau)$, we can allow ourselves to apply this nonoptimal algorithm to a random sample of the input set of size $\Theta(n/\tau)$. It turns out that this yields suitable estimates of multiplicities somewhat larger than τ, say, at least $\tau^{3/2}$. On the other hand, very small color classes are likely not to be represented in the sample at all, so that their sizes must be estimated in a different way. We here use the scattering in time described above, which enables us to estimate multiplicities up to roughly τ. Finally, in order to bridge the gap between τ and $\tau^{3/2}$, we use another scattering in time, but this time applied to a random sample of the input set of size $\Theta(n/\sqrt{\tau})$. The complete algorithm hence consists of three essentially independent subalgorithms, each of which "caters to" a different range of multiplicities.

LEMMA 8.7. *There is a constant $\varepsilon > 0$ such that for all given n, $\tau \in \mathbb{N}$ with $\tau \geqslant \log^* n$, the following problem can be solved on a* TOLERANT PRAM *using $O(\tau)$ time, $\lceil n/\tau \rceil$ processors and $O(n)$ space with probability at least $1 - 2^{-n^\varepsilon}$ (Monte Carlo): Given n integers $x_1, ..., x_n$ in the range $0..n$, compute independent nonnegative integer random variables $\hat{b}_1, ..., \hat{b}_n$ such that*

(A) $\sum_{i=1}^n \hat{b}_i = O(n)$;

(B) *For $i = 1, ..., n$ and for all $a \geqslant 1$, $\Pr(b_i > a\hat{b}_i) \leqslant 2^{-b_i/(8\tau)} + 2^{-2a}$;*

(C) *For $i = 1, ..., n$, $\Pr(b_i > \sqrt{\tau}\,\hat{b}_i) \leqslant 2^{-\sqrt{\tau}}$,*

where $b_i = |\{j: 1 \leqslant j \leqslant n \text{ and } x_j = i\}|$, for $i = 1, ..., n$.

Proof. Without loss of generality, we can assume that $2 \leqslant \tau \leqslant n^{1/2}$, since otherwise the problem is easily solved using the algorithm of Lemma 2.9. Let $K = 2^8$ and carry out the following algorithm:

Step 1. Apply one-pass scattering in time with phase count τ to the primary input $X_1, ..., X_n$ and the (trivial) profile input $8K, ..., 8K$ and let $\mathscr{X}'$ be the resulting set of noncolliding elements. For $i = 1, ..., n$, take $\hat{b}_i^{(1)} = 4\,|\mathscr{B}_i \cap \mathscr{X}'|$ (since $8K = \tau^{O(1)}$, we argued above that this quantity is readily available).

Step 2. Draw a random sample $\mathscr{Y} \subseteq \mathscr{X}$ by including each input element in $\mathscr{Y}$ independently of other elements and with probability $1/\lceil \sqrt{\tau} \rceil$. Repeat Step 1, but this time include only elements of $\mathscr{Y}$ in the primary input (i.e., replace each element not in $\mathscr{Y}$ by a dummy element with a value of 0), and let $\mathscr{Y}'$ be the resulting set of noncolliding elements. For $i = 1, ..., n$, take $\hat{b}_i^{(2)} = 8\lceil \sqrt{\tau} \rceil\,|\mathscr{B}_i \cap \mathscr{Y}'|$.

Step 3. Draw a random sample $\mathscr{Z} \subseteq \mathscr{X}$ by including each input element in $\mathscr{Z}$ independently of other elements and with probability $1/\tau$. By Chernoff bound (a), we can assume that $|\mathscr{Z}| = O(n/\tau)$. Use the algorithms of Lemma 8.1 and Theorem 4.6 to store $\mathscr{Z}$ in an array of size $O(n/\tau)$ and to replace the values of the elements in $\mathscr{Z}$ by values in a range of size $O(n/\tau)$. Then apply the algorithm of Lemma 8.3 to $\mathscr{Z}$ to obtain a profile $\hat{b}_1^{\mathscr{Z}}, ..., \hat{b}_n^{\mathscr{Z}}$ (take $\hat{b}_i^{\mathscr{Z}} = 0$ for each $i \in \{1, ..., n\}$ with $\mathscr{B}_i \cap \mathscr{Z} = \varnothing$). For $i = 1, ..., n$, let $\hat{b}_i^{(3)} = 4\tau\hat{b}_i^{\mathscr{Z}}$.

Step 4. For $i = 1, ..., n$, compute the final estimate of b_i as $\hat{b}_i = \max\{\hat{b}_i^{(1)}, \hat{b}_i^{(2)}, \hat{b}_i^{(3)}, K\}$.

For $i = 1, ..., n$, let $b_i^{\mathscr{Y}} = |\mathscr{B}_i \cap \mathscr{Y}|$ and $b_i^{\mathscr{Z}} = |\mathscr{B}_i \cap \mathscr{Z}|$. We can assume that $\hat{b}_1^{\mathscr{Z}}, ..., \hat{b}_n^{\mathscr{Z}}$ is indeed a coarse-profile for (the sequence of values of elements in) $\mathscr{Z}$. It is easy to see that with high probability, the resource requirements of the algorithm are as stated in the lemma. The correctness of the algorithm is demonstrated in the lemmas below, each of which shows one of the properties (A)–(C).

LEMMA 8.8. *With high probability, $\sum_{i=1}^n \hat{b}_i = O(n)$.*

Proof. $\sum_{i=1}^n \hat{b}_i^{(1)} = 4\sum_{i=1}^n |\mathscr{B}_i \cap \mathscr{X}'| \leqslant 4\sum_{i=1}^n b_i = 4n$. In the same way, $\sum_{i=1}^n \hat{b}_i^{(2)} \leqslant 8\lceil \sqrt{\tau} \rceil\,|\mathscr{Y}|$, and $|\mathscr{Y}| = O(n/\sqrt{\tau})$ with high probability by Chernoff bound (a). Finally, by the linear-sum condition, $\sum_{i=1}^n \hat{b}_i^{(3)} = 4\tau\sum_{i=1}^n \hat{b}_i^{\mathscr{Z}} = O(\tau\,|\mathscr{Z}|) = O(n)$. ∎

LEMMA 8.9. *For $i = 1, ..., n$ and for all $a \geqslant 1$, $\Pr(b_i > a\hat{b}_i) \leqslant 2^{-b_i/(8\tau)} + 2^{-2a}$.*

Proof. Clearly, $b_i^{\mathscr{Z}}$ is binomially distributed with expected value b_i/τ. Hence by Chernoff bound (b), $\Pr(b_i^{\mathscr{Z}} < b_i/(2\tau)) \leqslant 2^{-b_i/(8\tau)}$. Furthermore, by property (B) of a coarse-profile, $\Pr(b_i^{\mathscr{Z}} > 2a\hat{b}_i^{\mathscr{Z}}) \leqslant 2^{-2a}$. But $b_i^{\mathscr{Z}} \geqslant b_i/(2\tau)$ and $b_i^{\mathscr{Z}} \leqslant 2a\hat{b}_i^{\mathscr{Z}}$ together imply $b_i \leqslant 2\tau b_i^{\mathscr{Z}} \leqslant 4a\tau\hat{b}_i^{\mathscr{Z}} = a\hat{b}_i^{(3)}$. Hence $\Pr(b_i > a\hat{b}_i) \leqslant \Pr(b_i > a\hat{b}_i^{(3)}) \leqslant 2^{-b_i/(8\tau)} + 2^{-2a}$. ∎

LEMMA 8.10. *For* $i = 1, ..., n$, $\Pr(b_i > \sqrt{\tau}\,\hat{b}_i) \leq 2^{-\sqrt{\tau}}$.

Proof. Without loss of generality assume that $b_i > K\sqrt{\tau}$. By the definition of $\hat{b}_i$, if $b_i > \sqrt{\tau}\,\hat{b}_i$, then $b_i > \sqrt{\tau}\,\hat{b}_i^{(l)}$, for $l = 1, 2, 3$, so that for each $i \in \{1, ..., n\}$ we can show the event $b_i > \sqrt{\tau}\,\hat{b}_i$ to be unlikely in any of three ways. Correspondingly, we consider three cases. If $\tau < 4$, Case 2 disappears, and Cases 1 and 3 overlap; the argument remains valid, however.

Case 1. $b_i \leq 2K\tau$. If $b_i > \sqrt{\tau}\,\hat{b}_i^{(1)}$, then $|\mathscr{B}_i \cap \mathscr{X}'| = \hat{b}_i^{(1)}/4 \leq b_i/(4\sqrt{\tau}) \leq b_i/2$, which implies that at least $b_i/2$ elements of $\mathscr{B}_i$ collide in Step 1. But since $b_i \leq 2K\tau$, the density of the scattering in time of $\mathscr{B}_i$ is $b_i/(8K\tau) \leq 1/4$; hence, by Lemma 3.6(b), the probability that at least $b_i/2$ elements of $\mathscr{B}_i$ collide is bounded by $e^{-\zeta}$, where $\zeta = (b_i/2)^2/(32b_i \cdot 1) = b_i/2^7 \geq K\sqrt{\tau}/2^7 = 2\sqrt{\tau}$, from which the claim follows. Note that we actually showed the stronger relation $\Pr(b_i \geq 4\sqrt{\tau}\,|\mathscr{B}_i \cap \mathscr{X}'|) \leq 2^{-2\sqrt{\tau}}$, which will be used in Case 2 below.

Case 2. $2K\tau < b_i \leq K\tau^{3/2}$. If $b_i > \sqrt{\tau}\,\hat{b}_i^{(2)}$, then $|\mathscr{B}_i \cap \mathscr{Y}'| = \hat{b}_i^{(2)}/(8\lceil\sqrt{\tau}\rceil) \leq b_i/(\sqrt{\tau} \cdot 8\lceil\sqrt{\tau}\rceil)$, which happens only if either $b_i'' \leq b_i/(2\lceil\sqrt{\tau}\rceil)$ (the sample is small) or $|\mathscr{B}_i \cap \mathscr{Y}'| \leq b_i''/(4\sqrt{\tau})$ (many elements collide). Since b_i'' is binomially distributed with expected value $b_i/\lceil\sqrt{\tau}\rceil$, Chernoff bound (b) implies that $\Pr(b_i'' \leq b_i/(2\lceil\sqrt{\tau}\rceil)) \leq e^{-b_i/(8\lceil\sqrt{\tau}\rceil)}$. On the other hand, we know from Case 1 that $\Pr(b_i'' \geq 4\sqrt{\tau}\,|\mathscr{B}_i \cap \mathscr{Y}'| \mid b_i'' \leq 2K\tau) \leq 2^{-2\sqrt{\tau}}$, and by Chernoff bound (a), $\Pr(b_i'' > 2K\tau) \leq \Pr(b_i'' > 2b_i/\lceil\sqrt{\tau}\rceil) \leq e^{-b_i/(3\lceil\sqrt{\tau}\rceil)}$. Using that $b_i/\lceil\sqrt{\tau}\rceil \geq (2K\tau)/(2\sqrt{\tau}) = K\sqrt{\tau}$, we finally obtain that $\Pr(b_i > \sqrt{\tau}\,\hat{b}_i^{(2)}) \leq e^{-b_i/(8\lceil\sqrt{\tau}\rceil)} + 2^{-2\sqrt{\tau}} + e^{-b_i/(3\lceil\sqrt{\tau}\rceil)} \leq 2^{-K\sqrt{\tau}/8} + 2^{-2\sqrt{\tau}} + 2^{-K\sqrt{\tau}/3} \leq 2^{-\sqrt{\tau}}$.

Case 3. $b_i > K\tau^{3/2}$. By Lemma 8.9, $\Pr(b_i > \sqrt{\tau}\,\hat{b}_i) \leq 2^{-b_i/(8\tau)} + 2^{-2\sqrt{\tau}} \leq 2^{-K\sqrt{\tau}/8} + 2^{-2\sqrt{\tau}} \leq 2^{-\sqrt{\tau}}$. ∎

THEOREM 8.11. *There is a constant* $\varepsilon > 0$ *such that for all given* $n, \tau \in \mathbb{N}$ *with* $\tau \geq \log^* n$, *n-color coarse-profiling problems of size n can be solved on a* TOLERANT PRAM *using* $O(\tau)$ *time,* $\lceil n/\tau\rceil$ *processors and* $O(n)$ *space with probability at least* $1 - 2^{-n^\varepsilon}$ *(Monte Carlo).*

Proof. Assume without loss of generality that $\tau \leq n^{1/4}$ and that $b_i \leq n^{1/8}$, for $i = 1, ..., n$. Begin by computing a profile $\hat{b}_1, ..., \hat{b}_n$ for $x_1, ..., x_n$ with the properties described in Lemma 8.7. Then apply two-pass scattering in time with phase count τ to the primary input $X_1, ..., X_n$ and the profile input $\hat{b}_1, ..., \hat{b}_n$. For $i = 1, ..., n$, call $\mathscr{B}_i$ *well-estimated* if $b_i \leq \sqrt{\tau}\,\hat{b}_i$, and call each element of $\mathscr{B}_i$ *good* if $\mathscr{B}_i$ is well-estimated, and *bad* otherwise. By property (C) of Lemma 8.7, the expected number of bad elements in $\bigcup_{i=1}^n \mathscr{B}_i$ is at most $n \cdot 2^{-\sqrt{\tau}}$, and a martingale argument shows the actual number of bad elements to be $O(n/\tau + n^{3/4}) = O(n/\tau)$ with high probability. It is easy to see that the probability that a good element collides in one pass of a scattering in time is at most $1/\sqrt{\tau}$. Since we actually execute two passes, the probability that a good element is unsuccessful is at most $1/\tau$, so a martingale argument shows that the number of (good or bad) unsuccessful elements is $O(n/\tau)$ with high probability. Let $\mathscr{X}'$ be the set of successful elements.

Consider the situation after the scattering in time. For $i = 1, ..., n$, call $\mathscr{B}_i$ *resolved* if every element of $\mathscr{B}_i$ was successful in the scattering in time, and $\hat{b}_i \leq 16\tau$. The total number of unsuccessful elements being $O(n/\tau)$, we can use inverted guarded writing as in the proof of Lemma 8.1 to determine the set of resolved color classes. The important observation is that if $\mathscr{B}_i$ is resolved, then we can compute b_i exactly as $|\mathscr{B}_i \cap \mathscr{X}'|$, for $i = 1, ..., n$. Hence for $i = 1, ..., n$, do the following: If $\mathscr{B}_i$ is resolved, replace (the estimate) $\hat{b}_i$ by (the exact value) b_i; otherwise replace $\hat{b}_i$ by $\max\{\hat{b}_i, 16\tau\}$. Since all except $O(n/\tau)$ color classes are resolved, these changes preserve the linear-sum condition.

We must finally show that $\Pr(b_i > a\hat{b}_i) \leq 2^{-a}$, for $i = 1, ..., n$ and for all $a \geq 1$. Since this is obvious if $\mathscr{B}_i$ is resolved, let us assume that this is not the case. Then, however, $\hat{b}_i \geq 16\tau$, so we can assume without loss of generality that $b_i > 16\tau a$. Now by property (B) of Lemma 8.7, $\Pr(b_i > a\hat{b}_i) \leq 2^{-b_i/(8\tau)} + 2^{-2a} \leq 2^{-2a} + 2^{-2a} \leq 2^{-a}$. ∎

9. SEMISORTING

We first define the semisorting problem precisely and then outline the rest of the section.

Informally, the m-color semisorting problem inputs n elements with values in the range $0..m$ (elements with a value of 0 being dummy elements) and places these in an output array of size $O(n)$ such that all elements with a given color (i.e., nonzero value) occur together, separated only by empty cells. As usual, we model the input as n integers $x_1, ..., x_n$ in the range $0..m$ and the output as n nonnegative integers $y_1, ..., y_n$, where y_j should be thought of as the position in the output array of the jth input element, for $j = 1, ..., n$. Condition (1) below means that distinct (real) elements are not placed in the same output cell, condition (2) says that no element of a different color intervenes between two elements of the same color in the output array, and condition (3) requires the output array to be of size $O(n)$.

DEFINITION. For all $n, m \in \mathbb{N}$, the *m-color semisorting* problem of size n is the following: Given n integers $x_1, ..., x_n$ in the range $0..m$, compute n nonnegative integers $y_1, ..., y_n$ such that

(1) For $1 \leq i < j \leq n$, if $x_i \neq 0$, then $y_i \neq y_j$;

(2) For all $i, j, k \in \{1, ..., n\}$, if $y_i < y_j < y_k$ and $x_i = x_k$, then $x_j = x_i$;

(3) $\max\{y_j : 1 \leq j \leq n\} = O(n)$.

The *m-color strong semisorting* problem is identical, except for the additional requirement (4) below, where $b_i = |\{j: 1 \leqslant j \leqslant n \text{ and } x_j = i\}|$, for $i = 1, ..., m$.

(4) For all $j, k \in \{1, ..., n\}$ and $i \in \{1, ..., m\}$, if $x_j = x_k = i$, then $|y_j - y_k| = O(b_i)$.

It is perhaps instructive to compare the definition of semisorting with that of complete linear colored compaction given in Section 6. Informally, complete linear colored compaction requires upper bounds $d_1, ..., d_m$ on the sizes of the color classes to be specified as part of the input, and the input elements are placed in arrays $A_1, ..., A_m$ of sizes $O(d_1), ..., O(d_m)$, respectively, each of which is indexed starting at 1. In order to use an algorithm for complete linear colored compaction to semisort, one could therefore first compute estimates $\hat{b}_1, ..., \hat{b}_m$ with $\hat{b}_i \geqslant b_i$, for $i = 1, ..., m$, but $\sum_{i=1}^{m} \hat{b}_i = O(n)$, then use the given compaction subroutine with limits $\hat{b}_1, ..., \hat{b}_m$, and finally place the arrays $A_1, ..., A_m$ together in a base array of size $O(n)$.

The present section culminates in a proof that n-color semisorting problems of size n can be solved in $O(\log^* n)$ time with optimal speedup (with high probability). As mentioned in the introduction, this leads to an algorithm with optimal speedup for computing n-color fine-profiles in $O(\log^* n)$ time (Corollary 10.5). On the other hand, our path to optimal semisorting takes us via two auxiliary profilers (i.e., algorithms that compute profiles) of different types, both of which are finally subsumed by the n-color fine-profiler. More precisely, we are going to first describe an algorithm that, guided by a coarse-profile, can semisort in $O(\log^* n)$ time (Lemma 9.1); this algorithm, however, uses n processors and hence is not optimal. As a corollary we obtain a nonoptimal profiler that, informally speaking, overestimates all multiplicities while still satisfying the linear-sum condition (Corollary 9.12). Scattering in time together with an optimal coarse-profiler allows us to derive from this nonoptimal profiler an optimal profiler with almost the same properties, except that a small fraction of elements may belong to "badly estimated" color classes (Lemma 9.13). Using this last result, another application of scattering in time together with the nonoptimal semisorting algorithm finally yields an optimal semisorting algorithm (Theorem 9.14).

As mentioned above, our first goal is to devise an n-processor semisorting algorithm. The basic idea is simply to use techniques similar to those of Section 6 to compact the elements of each color class into an array of suitable size. However, this approach meets with major complications. Multiplicities must be estimated using a coarse-profiling algorithm, which means that the estimates obtained are not very reliable. Arrays must be allocated as described in Section 7, and colors cannot be handled independently, as far as the placement in arrays is concerned, since the failure probability for small color classes cannot be ignored. Instead it is necessary to monitor the progress of the colors throughout the process, pushing more resources towards colors that are not keeping pace with the rest.

LEMMA 9.1. *There is a constant $\varepsilon > 0$ such that for all given $n \in \mathbb{N}$, n-color semisorting problems of size n can be solved on a* TOLERANT PRAM *using $O(\log^* n)$ time, n processors and $O(n)$ space with probability at least $1 - 2^{-n^{\varepsilon}}$ (Las Vegas).*

Proof. In order to let the basic idea stand out clearly, we first describe and analyze a simplified algorithm that ignores a number of complications, and afterwards motivate the various bells and whistles that have to be added to the algorithm to actually make it work.

In the following the word "element" will be used exclusively to denote elements of color classes. The simplified algorithm is quite similar to the first part of the algorithm given in the proof of Theorem 6.2, the crucial difference between the two settings being that now we have to tolerate n color classes, rather than $\Theta(n^{1-\delta})$, for $\delta > 0$. We begin by computing a strictly increasing sequence $v_1, ..., v_T$ of positive integers similar to the sequence of successive values of v used in the proof of Theorem 6.2 (the exact requirements will be specified below), and then execute the following:

```
Let all elements be active;
for t := 1 to T do
  for each color class 𝒞_i pardo
    begin
      Allocate an array A_{i,t} of size 6b_i/v_t^3 to 𝒞_i;
      v_t-compact the active elements in 𝒞_i to A_{i,t},
        deactivating every lucky element in 𝒞_i;
    end;
```

The algorithm hence consists of T stages, each of which attempts to place the elements of each color class in an array of suitable size. The elements that are successfully placed in the array become inactive and do not participate in subsequent stages. Conceptually, if the algorithm succeeds in deactivating all elements, the elements of each color class $\mathscr{C}_i$ afterwards are stored in T arrays $A_{i,1}, ..., A_{i,T}$. In actual fact, there is no need to preserve $A_{i,t}$ beyond the end of Stage t, for $i = 1, ..., n$ and $t = 1, ..., T$, and the algorithm can reclaim the space allocated in each stage for reuse in the following stage. Instead, in Stage t, the algorithm remembers the total size $\sum_{l=1}^{t-1} |A_{i,l}|$ of the arrays allocated to $\mathscr{C}_i$ in earlier stages, for $i = 1, ..., n$ and $t = 1, ..., T$, and each element of $\mathscr{C}_i$ deactivated in Stage t adds this offset to its position in $A_{i,t}$ and stores the resulting absolute position. Informally, this has the effect of gluing the array $A_{i,t}$ onto the right end of an array already containing $A_{i,1}, ..., A_{i,t-1}$. After the last stage, it is therefore an easy matter to use the algorithm of Theorem 7.1 to allocate a single array A_i of size

$|A_{i,1}| + \cdots + |A_{i,T}|$ to $\mathscr{B}_i$ and to place the elements of $\mathscr{B}_i$ in A_i (informally, to move the elements of $\mathscr{B}_i$ from $A_{i,1}, \ldots, A_{i,T}$ to A_i). Provided that $\sum_{i=1}^{n} |A_i| = O(n)$, this produces a solution to the semisorting problem. In other words, if all elements are deactivated, the correctness of the algorithm will be guaranteed if we can show that $\sum_{i=1}^{n} \sum_{t=1}^{T} |A_{i,t}| = O(n)$. In the idealized algorithm above, this condition is satisfied, since $\sum_{i=1}^{n} \sum_{t=1}^{T} |A_{i,t}| = \sum_{t=1}^{T} \sum_{i=1}^{n} (6b_i/v_t^3) \leqslant \sum_{t=1}^{T} (6n/v_t^3) = O(n)$.

In order to analyze the rate with which elements are deactivated, fix $i \in \{1, \ldots, n\}$, let $t \in \{1, \ldots, T\}$ and assume by way of induction that the number of active elements in $\mathscr{B}_i$ has decreased to at most b_i/v_t^4 before Stage t. Then $\mathscr{B}_i$ is well-supplied in the v_t-compaction in Stage t, so that a fixed element in $\mathscr{B}_i$ remains active with probability at most 2^{-v_t} (by Lemma 6.1(a)). Therefore the expected number of elements in $\mathscr{B}_i$ that remain active is at most $b_i \cdot 2^{-v_t}$, which, for a suitable choice of the sequence $v_1, \ldots, v_T$, is significantly smaller than the b_i/v_{t+1}^4 required for the induction.

We now proceed to discuss the problems with the algorithm above. One such problem is that color classes may be too small to exhibit a "reliable" behavior, in a statistical sense. E.g., in the analysis in the preceding paragraph, even though the expected number of elements in $\mathscr{B}_i$ that remain active is significantly below b_i/v_{t+1}^4, the probability that their actual number exceeds b_i/v_{t+1}^4 may not be negligible (cf. Lemma 6.1(b), which yields little unless z is much larger than v). We counter this problem by treating small color classes specially; in particular, the space allocated to small color classes is larger, relative to their sizes, than for other color classes.

More significantly, we do not know the multiplicities $b_1, \ldots, b_n$, so we have to resort to estimates $\hat{b}_1, \ldots, \hat{b}_n$. One consequence of this is that we do not really know whether a color class is small; the color classes that are treated specially, as mentioned in the previous paragraph, are hence those whose estimates let them "appear" small.

Another difficulty is posed by the allocation of space to color classes. Since we intend to execute $\Theta(\log^* n)$ successive stages in a total of $O(\log^* n)$ time, we cannot carry out the allocation using the algorithm of Theorem 7.1, but have to resort to the incomplete allocation of Lemma 7.3 (the color classes themselves will be requesting elements in the sense of Lemma 7.3; however, recall our convention in this section to use the word "element" only for elements of color classes). As a consequence of the incomplete allocation, in each stage certain color classes will be unlucky (recall that this means that they do not receive the resources that they requested), so that they cannot participate in the v_t-compaction; this adds another complication to the analysis. Furthermore, in order for the resource requirements of the incomplete allocation to remain $O(n)$ in spite of the increase in v_t over the stages, it is necessary to ensure that the number of requests per stage decreases over the course of the execution. We therefore initially let all nonempty color classes be active, declare a color class to become inactive when it loses its last active element, and allocate space only to active color classes. Since the number of active color classes cannot exceed the number of active elements, the number of active color classes will decrease as required, provided that the number of active elements does so.

Finally, although for most color classes the incomplete compaction in a particular stage will succeed in deactivating most elements in the color class, for some color classes almost all elements may remain active; in particular, this surely happens for color classes that are unlucky in the stage under consideration, and it is likely to happen for those whose sizes were heavily underestimated. Given the algorithm as described so far, the problem will be aggravated over successive stages, since the available space decreases. In such cases we need to resort to an "emergency escape," which will be to compact into an array of size $6v_t^2\hat{b}_i$, rather than $6\hat{b}_i/v_t^3$. Note that we certainly cannot use $6v_t^2\hat{b}_i$ space always, since the total space requirements would be superlinear. In order to decide when to apply the emergency escape, we begin each stage with a "test scattering" for each active color class, except those that appear small. The test scattering is simply a conditional scattering by the active elements in the color class, with parameters chosen to allow color classes that are in need of the emergency escape to be roughly distinguished from those that are not; the actual bit observed is whether or not the fullness of the scattering equals 1.

Before describing the algorithm proper, we define the sequence $v_1, \ldots, v_T$ and collect in Lemma 9.2 below those of its properties that will be needed later. We can no longer define $v_1, \ldots, v_T$ simply as a sequence of "towers of two," as in the proofs of Theorem 6.2 and Lemma 8.3, since our analysis requires a somewhat smaller gap between consecutive elements in the sequence. The basic idea remains the same, however, and we will still have $T = O(\log^* n)$.

Briefly let $f(z) = z - 44 \log z$, for $z > 0$. Then $f(2) < 0$, $f(2^9) > 0$, $f(z) \to \infty$ for $z \to \infty$, and f' has only one zero, so that $z > 44 \log z \geqslant 22\lceil \log z \rceil$ for all $z \geqslant 2^9$. The algorithm below hence outputs a finite sequence, which we take to be $v_T, \ldots, v_1$ (note the reverse indexing).

```
z := ⌈n^{1/88}⌉;
repeat
    write(z);
    z := 22⌈log z⌉;
until z < 2^{15};
```

Clearly $v_1 < 2^{2^{15}} = O(1)$. We will assume without loss of generality that $T \geqslant 2$ (otherwise n is bounded by a constant). Then $2^{15} \leqslant v_1 < v_2 < \cdots < v_T = \lceil n^{1/88} \rceil$. For a sufficiently

large constant z_0, clearly $22\lceil\log(22\lceil\log z\rceil)\rceil \leq 44\log(44\log z) = 44(\log(44) + \log\log z) \leq \log z$ for all $z \geq z_0$. Hence $T \leq 2\log^* n + O(1)$.

LEMMA 9.2. (a) *For* $t = 1, \ldots, T-1$, $v_{t+1}^{22} \leq 2^{v_t} \leq v_{t+1}^{44}$;

(b) *For* $t = 1, \ldots, T-1$, $v_t^{22} \leq v_{t+1}$.

Proof. (a) $\log(v_{t+1}^{22}) = 22\log v_{t+1} \leq 22\lceil\log v_{t+1}\rceil = v_t \leq 44\log v_{t+1} = \log(v_{t+1}^{44})$, from which the relation follows by exponentiation.

(b) We noted above that $2^z \geq z^{44}$ for $z \geq 2^9$. Since $v_t \geq 44 \cdot 2^9$, part (a) therefore implies that $v_{t+1} \geq 2^{v_t/44} \geq (v_t/44)^{44} \geq v_t^{22}$. ∎

The algorithm begins by computing a coarse-profile $\tilde{b}_1, \ldots, \tilde{b}_n$ for $x_1, \ldots, x_n$. By Theorem 8.2, we can assume that $\tilde{b}_i$ is a linear overestimate for b_i for all $i \in I$, where I is a known subset of $\{1, \ldots, n\}$ with the property that $i \in I$ for all colors i with $b_i \geq n^{1/88}$. Since we can clearly remove all colors i with $\tilde{b}_i < n^{1/88}$ from I without affecting the property of I just mentioned, we can also assume that $|I| = O(n^{87/88})$. We want to apply (the nonoptimal part of) the algorithm of Theorem 6.2 to place the elements of $\mathscr{B}_i$ in an array of size $O(b_i)$ (with $\tilde{b}_i$ serving as the limit for $\mathscr{B}_i$), for all $i \in I$. Recall that the algorithm of Theorem 6.2 can cope with up to $\Theta(n^{1-\delta})$ colors, for arbitrary fixed $\delta > 0$, as long as we provide a means of performing the necessary space allocation. Since Theorem 7.1 is now available, the latter condition no longer is a problem. Using the algorithms of Theorem 4.6 and Lemma 8.1 to replace the colors in I by colors in a range of size $O(n^{87/88})$, we can therefore apply the algorithm of Theorem 6.2 as stated. This preprocessing serves to let us assume without loss of generality that $b_i \leq n^{1/88}$, for $i = 1, \ldots, n$. We want to work with estimates that are at least 1 and at most $n^{1/88}$ and therefore take $\hat{b}_i = \min\{\max\{\tilde{b}_i, 1\}, \lfloor n^{1/88}\rfloor\}$, for $i = 1, \ldots, n$. It is easy to see that $\hat{b}_1, \ldots, \hat{b}_n$ is still a coarse-profile for $x_1, \ldots, x_n$; in particular, $\hat{b}_1, \ldots, \hat{b}_n$ are independent random variables. Additionally, $b_i \leq v_T\hat{b}_i$ and $\hat{b}_i \leq n^{1/88}$, for $i = 1, \ldots, n$. The algorithm now computes the sequence $v_T, \ldots, v_1$, which can obviously be done in $O(\log^* n)$ time, and proceeds as follows:

```
(1)  Let all elements and all nonempty color classes be
     active;
(2)  for t := 1 to T do (* Stage t *)
(3)    for each active color class 𝒜_i pardo
(4)      begin
(5)        if b̂_i ≤ v_t^17
(6)          then (* apparently-small *) Size_{i,t} := 6v_t^19
             else
(7)          begin (* base Size_{i,t} on test scattering *)
(8)            v_t-allocate v_t memory cells and v_t
               processors to 𝒜_i;
(9)            if 𝒜_i was unlucky in the allocation in
               line (8)
(10)             then goto line (21);
(11)           Let the elements in 𝒜_i carry out a condi-
               tional scattering 𝒮_i with probability v_t^7/b̂_i
               and of width v_t;
(12)           if 𝒮_i has fullness 1
(13)             then (* apparently-huge *) Size_{i,t} :=
                 6v_t^2 b̂_i
(14)             else (* normal case *) Size_{i,t} := ⌈6b̂_i/v_t^3⌉;
(15)         end;
(16)       v_t-allocate an array A_{i,t} of size Size_{i,t} to 𝒜_i;
(17)       if 𝒜_i was lucky in the allocation in line (16)
(18)         then v_t-compact the active elements in 𝒜_i to
             A_{i,t}, deactivating every lucky element in 𝒜_i;
(19)       if no element in 𝒜_i remains active
(20)         then make 𝒜_i inactive;
(21)     end;
```

For $t = 1, \ldots, T$, let *Stage* t be the tth execution of lines (3)–(21) and say that a color class is active in Stage t if it is active at the beginning of Stage t and that it is unlucky in Stage t if it is unlucky in the incomplete allocation in either line (8) or line (16) in Stage t. A color class that is unlucky in some stage drops out of that stage and rejoins the computation in the beginning of the next stage, if any; this is realized via a goto instruction in line (10) and a conditional instruction in line (17).

The goal of the analysis is to show that with high probability, the algorithm deactivates all elements using $O(n)$ processors and $O(n)$ space and allocating arrays $A_{i,t}$ of total size $O(n)$. Since a stage can be executed in constant time, the algorithm is then correct and its resource requirements are as claimed in Lemma 9.1. A key property established below is that the number of active elements (and hence of active color classes) decreases rapidly over the execution of the algorithm. More precisely, we will show (Lemma 9.8) that with high probability the number of elements active at the beginning of Stage t, for $t = 1, \ldots, T$, is $O(n/v_t^{22})$. The proof of this key property consists of two main parts. We first identify certain favorable conditions that may apply to a color class in a stage, show that these conditions together imply that the color class is well-supplied, in the sense of Lemma 6.1, in the v_t-compaction in the given stage (Lemma 9.3), and note that for well-supplied color classes the rate of deactivation is essentially as in the idealized analysis of the simplified algorithm earlier in this section (Lemma 9.6). We then show that only very few color classes lack the favorable conditions (Lemma 9.7).

For $i = 1, \ldots, n$ and $t = 1, \ldots, T$, denote by $N_{i,t}$ the number of active elements in $\mathscr{B}_i$ at the start of Stage t. $\mathscr{B}_i$ is said to be *apparently-small* in Stage t if $\hat{b}_i \leq v_t^{17}$, and to be *apparently-huge* in Stage t if a test scattering for $\mathscr{B}_i$ is carried out in Stage t and achieves a fullness of 1. $\mathscr{B}_i$ is *well-estimated* in Stage t if

$b_i \leq v_t \hat{b}_i$ (i.e., the size of $\mathscr{B}_i$ may have been underestimated, but at most by a factor of v_t), and $\mathscr{B}_i$ is *well-supplied* in Stage t if it is active in Stage t, not unlucky in any of the incomplete allocations in lines (8) and (16) in Stage t, and well-supplied in the incomplete compaction in line (18) in Stage t. Recalling the definition of "well-supplied" in Section 6, we observe that $\mathscr{B}_i$ is well-supplied in Stage t if and only if an array $A_{i,t}$ of size at least $6v_t N_{i,t}$ is allocated to $\mathscr{B}_i$ in Stage t. A sufficient set of conditions for this to happen is formulated in the following lemma.

LEMMA 9.3. *For $i = 1, ..., n$ and $t = 1, ..., T$, $\mathscr{B}_i$ is well-supplied in Stage t if it is active in Stage t and each of the following conditions holds:*

(1) *$\mathscr{B}_i$ is well-estimated in Stage t;*

(2) *$\mathscr{B}_i$ is not unlucky (in any of the v_t-allocation steps) in Stage t;*

(3) *$\mathscr{B}_i$ is apparently-small or apparently-huge in Stage t, or $N_{i,t} \leq \hat{b}_i / v_t^4$.*

Proof. Let $i \in \{1, ..., n\}$ and $t \in \{1, ..., T\}$ and assume that $\mathscr{B}_i$ is active in Stage t and that conditions (1)–(3) hold. In particular, an array $A_{i,t}$ of size $Size_{i,t}$ is allocated to $\mathscr{B}_i$ (condition (2)). If $\mathscr{B}_i$ is apparently-small in Stage t, then $b_i \leq v_t \hat{b}_i \leq v_t^{18}$ (condition (1)) and $Size_{i,t} = 6v_t^{19}$. Otherwise $Size_{i,t} \geq 6\hat{b}_i / v_t^3$, and if $N_{i,t} > \hat{b}_i / v_t^4$, then $Size_{i,t} = 6v_t^2 \hat{b}_i \geq 6v_t b_i$ (conditions (3) and (1)). In all cases $Size_{i,t} \geq 6v_t N_{i,t}$, i.e., $\mathscr{B}_i$ is well-supplied in Stage t. ∎

Recall that the density of a conditional scattering with probability p and of width s carried out by a set of m elements is defined as mp/s.

LEMMA 9.4. *For $t = 1, ..., T$, if a test scattering is executed in Stage t by a color class $\mathscr{B}_i$ with $N_{i,t} > \hat{b}_i / v_t^4$, then the probability that $\mathscr{B}_i$ does not become apparently-huge in Stage t is at most 2^{-v_t}.*

Proof. The density of the test scattering is at least $(\hat{b}_i / v_t^4) \cdot (v_t^7 / \hat{b}_i) \cdot (1/v_t) = v_t^2$. Hence by Lemma 3.1(c), the probability in question is at most $v_t \cdot 2^{-v_t^2} \leq 2^{-v_t}$. ∎

LEMMA 9.5. *With high probability, the algorithm deactivates all elements.*

Proof. We first show that with high probability, conditions (1)–(3) of Lemma 9.3 are satisfied in Stage T for all active color classes. We already noted that $b_i \leq v_T \hat{b}_i$, for $i = 1, ..., n$, so that every color class is well-estimated in Stage T, i.e., condition (1) is satisfied. By Lemma 7.3(a), the probability that some active color class is unlucky in Stage T is at most $2n \cdot 2^{-v_T}$; i.e., condition (2) is also satisfied with high probability. Condition (3), finally, follows directly from Lemma 9.4.

By what was shown above and Lemma 9.3, with high probability every active color class is well-supplied in Stage T. Lemma 6.1(a) implies that the probability that a fixed active element in an active and well-supplied color class is unlucky in the v_T-compaction in Stage T is at most 2^{-v_T}. Hence with high probability, no element remains active at the end of Stage T. ∎

LEMMA 9.6. *For $t = 1, ..., T-1$, with high probability the number of active elements in well-supplied color classes at the end of Stage t is $O(n / v_{t+1}^{22})$.*

Proof. An element whose color class is well-supplied in Stage t remains active at the end of Stage t only if it is unlucky in the v_t-compaction in Stage t. However, Lemmas 6.1(b) and 9.2 show the number of such elements to be no larger than n / v_{t+1}^{22}, except with probability $2e^{-\zeta}$, where $\zeta \geq (n / v_{t+1}^{22})^2 / (2^9 n v_t^3) \geq n / (2^9 v_{t+1}^{45}) = \Omega(n^{1/4})$, i.e., except with negligible probability. ∎

Lemma 9.6 shows that the number of elements in well-supplied color classes decreases as required. In Lemma 9.7 we prove that the elements in color classes that are active but not well-supplied are so few that they can be ignored in this context. Informally, the reason for this is that if an active color class is not well-supplied in Stage t, then either it is unlucky, or its estimate is off by a factor of more than v_t, or the test scattering for the color class does not achieve fullness 1 although its density is at least v_t^2, all of which are unlikely.

LEMMA 9.7. *Let $t \in \{1, ..., T-1\}$ and take $I = \{i: 1 \leq i \leq n$ and $\mathscr{B}_i$ is active but not well-supplied in Stage $t\}$. Then, with high probability, $\sum_{i \in I} (b_i + \hat{b}_i) = O(n / v_{t+1}^{22})$.*

Proof. If a color class $\mathscr{B}_i$ is active but not well-supplied in Stage t, then one of Conditions (1)–(3) of Lemma 9.3 must be violated. Therefore the index sets I', I'', and I''' defined below cover all of I, i.e., $I' \cup I'' \cup I''' = I$; we will show that with high probability the sum $\sum_i (b_i + \hat{b}_i)$ over each of these index sets is $O(n / v_{t+1}^{22})$.

$$I' = \{i \in I : \mathscr{B}_i \text{ is not well-estimated in Stage } t\},$$

$$I'' = \{i \in I : \mathscr{B}_i \text{ is unlucky in Stage } t\}, \quad \text{and}$$

$$I''' = \{i \in I \backslash (I' \cup I'') : \mathscr{B}_i \text{ is neither apparently-small nor apparently-huge in Stage } t \text{ and } N_{i,t} > \hat{b}_i / v_t^4\}.$$

By definition, if $i \in I'$, then $\hat{b}_i \leq b_i / v_t$, which, by property (B) of a coarse-profile, happens with probability at most 2^{-v_t}. A martingale argument and Lemma 9.2(a) then shows that with high probability, $\sum_{i \in I'} (b_i + \hat{b}_i) \leq 2 \sum_{i \in I'} b_i = O(n / 2^{v_t} + n^{1/88} n^{5/8}) = O(n / v_{t+1}^{22})$. In the rest of the proof we consider all random choices made by the algorithm in Stages $1, ..., t-1$ to be fixed in an arbitrary manner. Write $I'' = I_1'' \cup I_2''$, where I_1'' and I_2'' are the residue sets of the v_t-allocations in lines (8) and (16), respectively. If we further define $S_1 = \sum_{i \in I_1''} (b_i + \hat{b}_i)$ and $S_2 = \sum_{i \in I_2''} (b_i + \hat{b}_i)$ as

functions of these residue sets, it is easy to see that both S_1 and S_2 satisfy a Lipschitz condition with a constant of size $O(n^{1/88})$ (recall that $b_i + \hat{b}_i = O(n^{1/88})$, for $i = 1, ..., n$). We also know for each v_t-allocation that a fixed color class is unlucky with probability at most 2^{-v_t} (Lemma 7.3(a)), so that $E(S_1 + S_2) = O(n/2^{v_t})$. By two straightforward applications of Lemma 7.3(c), we thus obtain that with high probability, $\sum_{i \in I'''} (b_i + \hat{b}_i) = S_1 + S_2 = O(n/2^{v_t} + n^{1/88} v_t^3 n^{5/8}) = O(n/v_{t+1}^{22} + n^{3/4}) = O(n/v_{t+1}^{22})$. Finally, if a color class $\mathscr{B}_i$ is neither apparently-small nor unlucky in Stage t, a test scattering is carried out for $\mathscr{B}_i$ in Stage t. Hence, by Lemma 9.4, $\Pr(i \in I''') \leqslant 2^{-v_t}$, for $i = 1, ..., n$, and therefore $E(\sum_{i \in I'''} (b_i + \hat{b}_i)) = O(\sum_{i=1}^n (b_i + \hat{b}_i)/2^{v_t}) = O(n/2^{v_t})$. A simple martingale argument now ensures that, with high probability, $\sum_{i \in I'''} (b_i + \hat{b}_i) = O(n/2^{v_t} + n^{3/4}) = O(n/v_{t+1}^{22})$. ∎

LEMMA 9.8. *For $t = 1, ..., T-1$, with high probability the number of elements (and hence color classes) active at the end of Stage t is $O(n/v_{t+1}^{22})$.*

Proof. Immediate from Lemmas 9.6 and 9.7. ∎

We finally show that the total size of the arrays $A_{i,t}$ allocated in Stage t is $O(n/v_t^3)$, for $t = 1, ..., T$ (Lemma 9.10), from which it will follow not only that the algorithm is correct, but also that it uses $O(n)$ processors and $O(n)$ space. Disregarding the arrays allocated to apparently-huge color classes, this can easily be done using Lemma 9.8. In order to handle the apparently-huge color classes, however, we first have to show the following technical lemma, which says that if a color class $\mathscr{B}_i$ is well-supplied in Stage t, then it is unlikely to contain more than $\max\{v_{t+1}\sqrt{b_i},\ b_i/v_{t+1}^8\}$ active elements at the beginning of Stage $t+1$.

LEMMA 9.9. *Let $t \in \{1, ..., T-1\}$ and take $I = \{i: 1 \leqslant i \leqslant n,\ \mathscr{B}_i$ is active and well-supplied in Stage t and $N_{i,t+1} > \max\{v_{t+1}\sqrt{b_i},\ b_i/v_{t+1}^8\}\}$. Then, with high probability, $\sum_{i \in I} (b_i + \hat{b}_i) = O(n/v_{t+1}^{22})$.*

Proof. Consider all random choices made in the algorithm before the v_t-compaction in line (18) in Stage t to be fixed in an arbitrary way and let $i \in \{1, ..., n\}$. Since $b_i/v_{t+1}^8 \geqslant b_i/2^{v_t}$, Lemmas 6.1(b) and 9.2(b) imply that if $\mathscr{B}_i$ is active and well-supplied in Stage t, then $N_{i,t+1} > \max\{v_{t+1}\sqrt{b_i},\ b_i/v_{t+1}^8\}$ with probability at most $2e^{-\zeta}$, where $\zeta = v_{t+1}^2 b_i/(2^9 b_i v_t^3) \geqslant v_t^{44}/(2^9 v_t^3) \geqslant 2v_t$. We have thus shown that $\Pr(i \in I) \leqslant 2 \cdot 2^{-2v_t} \leqslant 2^{-v_t}$. Similarly as in the proof of Lemma 9.7, let $S = \sum_{i \in I} (b_i + \hat{b}_i)$ and note that S satisfies a Lipschitz condition with a constant of size $O(n^{1/88})$. Now $E(S) \leqslant \sum_{i=1}^n (b_i + \hat{b}_i) \cdot 2^{-v_t} = O(n \cdot 2^{-v_t}) = O(n/v_{t+1}^{22})$, and by Lemma 6.1(c), with high probability $S = O(E(S) + n^{1/88} v_t^3 n^{5/8}) = O(n/v_{t+1}^{22} + n^{3/4}) = O(n/v_{t+1}^{22})$. ∎

LEMMA 9.10. *For $t = 1, ..., T$, the total size of the arrays $A_{i,t}$ allocated in Stage t is $O(n/v_t^3)$.*

Proof. The claim is obvious for $t = 1$ since $v_1 = O(1)$, so fix $t \in \{2, ..., T\}$. By Lemma 9.8, with high probability the total number of arrays allocated in Stage t is $O(n/v_t^{22})$. Hence the total size of the arrays allocated in Stage t to color classes that are not apparently-huge is

$$O\left((n/v_t^{22}) \cdot (v_t^{19} + 1) + \sum_{i=1}^n (\hat{b}_i/v_t^3)\right) = O(n/v_t^3),$$

as desired.

What remains is to bound the total size of the arrays allocated to apparently-huge color classes. Let $I = \{i: 1 \leqslant i \leqslant n$ and $\mathscr{B}_i$ is apparently-huge in Stage $t\}$. It suffices to show that $\sum_{i \in I} \hat{b}_i = O(n/v_t^5)$ with high probability, since then the total size of the arrays allocated to apparently-huge color classes in Stage t is $O(\sum_{i \in I} v_t^2 \hat{b}_i) = O(n/v_t^3)$. To this end we partition I into three subsets:

$$I' = \{i \in I: \mathscr{B}_i \text{ is not well-supplied in Stage } t-1 \text{ or } \mathscr{B}_i \text{ is not well-estimated in Stage } t\},$$

$$I'' = \{i \in I \backslash I': N_{i,t} > \hat{b}_i/v_t^7\}, \text{ and}$$

$$I''' = \{i \in I \backslash I': N_{i,t} \leqslant \hat{b}_i/v_t^7\}.$$

Using property (B) of a coarse-profile as in the proof of Lemma 9.7 as well as Lemma 9.7 itself, we find that with high probability, $\sum_{i \in I'} \hat{b}_i = O(n/v_t^{22} + n/2^{v_t} + n^{3/4}) = O(n/v_t^5)$. Suppose next that $i \in I''$. Then $N_{i,t} > \hat{b}_i/v_t^7$ and $\mathscr{B}_i$ is well-estimated in Stage t, i.e., $\hat{b}_i \geqslant b_i/v_t$. Also, since $\mathscr{B}_i$ is apparently-huge in Stage t, it cannot be apparently-small in Stage t, so $\hat{b}_i > v_t^{17}$. It follows that $N_{i,t} \geqslant b_i/v_t^8$ and also that $N_{i,t} > \hat{b}_i^{1/2} v_t^{17/2}/v_t^7 \geqslant \sqrt{b_i/v_t} \cdot v_t^{3/2} = v_t\sqrt{b_i}$. But then, by Lemma 9.9, $\sum_{i \in I''} \hat{b}_i = O(n/v_t^5)$ with high probability.

As concerns I''', finally, we use the fact that a color class $\mathscr{B}_i$ with $N_{i,t} \leqslant \hat{b}_i/v_t^7$ is very unlikely to become apparently-huge in Stage t. Specifically, according to Lemma 3.1(d), the probability of this event is at most

$$\left(\frac{\hat{b}_i}{v_t^7} \cdot \frac{v_t^7}{\hat{b}_i} \cdot e \cdot \frac{1}{v_t}\right)^{v_t} \leqslant 2^{-v_t},$$

and a simple martingale argument shows that $\sum_{i \in I'''} \hat{b}_i = O(n/v_t^5 + n^{1/88} n^{3/4}) = O(n/v_t^5)$ with high probability. ∎

LEMMA 9.11. *With high probability, the algorithm is correct and uses $O(n)$ processors and $O(n)$ space.*

Proof. We have already argued that the correctness of the algorithm follows from Lemmas 9.5 and 9.10. It uses $O(n)$ processors and $O(n)$ space, plus the resources needed for the incomplete allocations in lines (8) and (16), which are $O(n)$ by Lemmas 9.8, 9.10 and 7.3, and the resources needed for the incomplete compaction in line (18), which

are $O(n)$ by Lemmas 9.8 and 6.1. This ends the proofs of Lemmas 9.11 and 9.1. ▮

COROLLARY 9.12. *There is a constant $\varepsilon > 0$ such that for all given $n \in \mathbb{N}$, the following problem can be solved on a* TOLERANT PRAM *using $O(\log^* n)$ time, n processors and $O(n)$ space with probability at least $1 - 2^{-n^\varepsilon}$ (Las Vegas): Given n integers $x_1, ..., x_n$ in the range $0..n$, compute n nonnegative integers $\hat{b}_1, ..., \hat{b}_n$ such that*

(A) $\sum_{i=1}^{n} \hat{b}_i = O(n)$;

(B) *For $i = 1, ..., n$, $\hat{b}_i \geq b_i$,*

where $b_i = |\{j: 1 \leq j \leq n \text{ and } x_j = i\}|$, for $i = 1, ..., n$.

Proof. After semisorting the input elements into an array A of size $O(n)$ using the algorithm of Lemma 9.1, we can use the algorithm of Lemma 2.8(a) to store them in a linked list in the order in which they occur in A. This makes it easy to compute the first and the last element in A of each nonempty color class, which identifies nonoverlapping subarrays $A_1, ..., A_n$ of A such that A_i contains all elements of $\mathscr{B}_i$, for $i = 1, ..., n$. All that remains is to take $\hat{b}_i = |A_i|$, for $i = 1, ..., n$. ▮

The final goal in this section is to take the step from the nonoptimal algorithm of Lemma 9.1 to an optimal semisorting algorithm. We first describe an algorithm with optimal speedup for computing a profile with the properties described in Corollary 9.12, except that condition (B) may be violated in a few cases.

LEMMA 9.13. *There is a constant $\varepsilon > 0$ such that for all given n, $\tau \in \mathbb{N}$ with $\tau \geq \log^* n$, the following problem can be solved on a* TOLERANT PRAM *using $O(\tau)$ time, $\lceil n/\tau \rceil$ processors and $O(n)$ space with probability at least $1 - 2^{-n^\varepsilon}$ (Monte Carlo): Given n integers $x_1, ..., x_n$ in the range $0..n$, compute n nonnegative integers $\tilde{b}_1, ..., \tilde{b}_n$ such that*

(A) $\sum_{i=1}^{n} \tilde{b}_i = O(n)$;

(B) $\sum_{i \in I} b_i = O(n/\tau)$,

where $b_i = |\{j: 1 \leq j \leq n \text{ and } x_j = i\}|$, for $i = 1, ..., n$, and $I = \{i: 1 \leq i \leq n \text{ and } b_i > \tilde{b}_i\}$.

Proof. Assume that $\tau \leq n^{1/4}$ and that $b_i \leq n^{1/8}$, for $i = 1, ..., n$. First use the algorithm of Theorem 8.11 to compute a coarse-profile $\hat{b}_1, ..., \hat{b}_n$ for $x_1, ..., x_n$. Then apply two-pass scattering in time with phase count τ to the primary input $X_1, ..., X_n$ and the profile input $\hat{b}_1, ..., \hat{b}_n$ and let $\mathscr{X}'$ and $\mathscr{X}''$ be the resulting sets of successful and unsuccessful elements, respectively. It follows almost exactly as in the proof of Theorem 8.11 that $|\mathscr{X}''| = O(n/\tau)$ with high probability. We can hence use the algorithms of Theorem 4.6 and Lemma 8.1 to store $\mathscr{X}''$ in an array of size $O(n/\tau)$ and to replace the values of elements in $\mathscr{X}''$ by values in a range of size $O(n/\tau)$, after which we can use the algorithm of Corollary 9.12 to compute a profile $\hat{b}_1'', ..., \hat{b}_n''$ such that $\sum_{i=1}^{n} \hat{b}_i'' = O(n/\tau)$, but $\hat{b}_i'' \geq |\mathscr{B}_i \cap \mathscr{X}''|$, for $i = 1, ..., n$ (take $\hat{b}_i'' = 0$ for each $i \in \{1, ..., n\}$ with $\mathscr{B}_i \cap \mathscr{X}'' = \emptyset$).

Now draw a random sample $\mathscr{Y}$ from $\mathscr{X}'$ (not from the full input set $\mathscr{X}$) by including each element of $\mathscr{X}'$ in $\mathscr{Y}$ with probability $1/\tau$ and independently of other elements. By Chernoff bound (a), with high probability $|\mathscr{Y}| = O(n/\tau)$. Exactly as described for $\mathscr{X}''$ above, we can compute a profile $\hat{b}_1^{\mathscr{Y}}, ..., \hat{b}_n^{\mathscr{Y}}$ such that $\sum_{i=1}^{n} \hat{b}_i^{\mathscr{Y}} = O(n/\tau)$, but $\hat{b}_i^{\mathscr{Y}} \geq |\mathscr{B}_i \cap \mathscr{Y}|$, for $i = 1, ..., n$. For $i = 1, ..., n$, let $b_i' = |\mathscr{B}_i \cap \mathscr{X}'|$, $b_i'' = |\mathscr{B}_i \cap \mathscr{X}''|$ and $b_i^{\mathscr{Y}} = |\mathscr{B}_i \cap \mathscr{Y}|$. For $i = 1, ..., n$, if $\hat{b}_i \leq \tau^2$, then we can compute b_i' exactly (cf. the description of scattering in time preceding Lemma 8.7) and take $\tilde{b}_i = b_i' + \hat{b}_i'' \geq b_i' + b_i'' = b_i$; otherwise take $\tilde{b}_i = \max\{2\tau \hat{b}_i^{\mathscr{Y}}, \tau^2\} + \hat{b}_i''$.

Property (A) is satisfied, since $\sum_{i=1}^{n} \tilde{b}_i \leq \sum_{i=1}^{n} (b_i' + 2\tau \hat{b}_i^{\mathscr{Y}} + \hat{b}_i'') + O(n) = O(|\mathscr{X}''| + 2\tau |\mathscr{Y}| + |\mathscr{X}''| + n) = O(n)$. As for property (B), fix $i \in \{1, ..., n\}$ and note first that we cannot have $b_i > \tilde{b}_i$ unless $b_i' > \max\{2\tau \hat{b}_i^{\mathscr{Y}}, \tau^2\}$. However, $b_i' > 2\tau \hat{b}_i^{\mathscr{Y}}$ implies $b_i^{\mathscr{Y}} < b_i'/(2\tau)$, which under the condition $b_i' > \tau^2$ happens with probability at most $e^{-b_i'/(8\tau)} \leq e^{-\tau/8}$, by Chernoff bound (b). The desired result now follows by a martingale argument. ▮

THEOREM 9.14. *There is a constant $\varepsilon > 0$ such that for all given n, $\tau \in \mathbb{N}$ with $\tau \geq \log^* n$, n-color semisorting problems of size n can be solved on a* TOLERANT PRAM *using $O(\tau)$ time, $\lceil n/\tau \rceil$ processors and $O(n)$ space with probability at least $1 - 2^{-n^\varepsilon}$ (Las Vegas).*

Proof. Observe first that it suffices to partition the input into two subsets and to semisort these into arrays Q' and Q'' of size $O(n)$ each. For then, as in the proof of Corollary 9.12, we can divide Q' into nonoverlapping subarrays $A_1', ..., A_n'$ and Q'' into nonoverlapping subarrays $A_1'', ..., A_n''$ such that each element of $\mathscr{B}_i$ is stored either in A_i' or in A_i'', for $i = 1, ..., n$, after which we can use the algorithm of Theorem 7.1 to allocate an array A_i of size $|A_i'| + |A_i''|$ to $\mathscr{B}_i$ from a base array of size $O(\sum_{i=1}^{n} (|A_i'| + |A_i''|)) = O(n)$ and store all elements of $\mathscr{B}_i$ in A_i, for $i = 1, ..., n$.

By this observation, it suffices to semisort the n input elements with a "waste" of $O(n/\tau)$ elements, i.e., with $O(n/\tau)$ elements not placed in the output array. This is because the elements that could not be placed are sufficiently few to be semisorted by the algorithm of Lemma 9.1 (following a compaction and renaming according to Theorem 4.6 and Lemma 8.1), after which we are in the situation described above.

As usual, assume that $\tau \leq n^{1/4}$ and that $b_i \leq n^{1/8}$, for $i = 1, ..., n$. The algorithm begins by computing a profile $\tilde{b}_1, ..., \tilde{b}_n$ for $x_1, ..., x_n$ with the properties described in Lemma 9.13, after which it applies one-pass scattering in time with phase count τ to the primary input $X_1, ..., X_n$ and the profile input $\tilde{b}_1, ..., \tilde{b}_n$. Similarly as in the proof of Theorem 8.11, call $\mathscr{B}_i$ *well-estimated* if $b_i \leq \tilde{b}_i$, and call each element of $\mathscr{B}_i$ *good* if $\mathscr{B}_i$ is well-estimated, and *bad* otherwise, for $i = 1, ..., n$. Our first source of "waste" are the bad

elements; by property (B) of Lemma 9.13, their number is $O(n/\tau)$ with high probability. A second source of "waste" are the colliding good elements. Since a good element collides with probability at most $1/\tau$, a martingale argument shows that the number of colliding good elements is also $O(n/\tau + n^{3/4}) = O(n/\tau)$ with high probability. A third and last source of "waste" will be good elements that cannot be placed in the output array although they did not collide. We now describe a procedure that uses the output of the scattering in time to semisort most of the noncolliding good elements.

Recall that scattering in time with profile input $\tilde{b}_1, ..., \tilde{b}_n$, as described in Section 8, uses arrays $A_1, ..., A_n$ of list headers and counters allocated within a base array of size $O(n)$, where $|A_i| = \tilde{b}_i$, for $i = 1, ..., n$. For $i = 1, ..., n$, divide A_i into $\lceil \tilde{b}_i/\tau \rceil$ *segments*, each of size at most τ, and say that an element is stored in a segment if it belongs to a list whose header is stored in (a cell in) the segment. Further take $b_i' = |\mathscr{B}_i \cap \mathscr{X}'|$, where $\mathscr{X}'$ is the set of noncolliding elements.

Now associate a *target array* with each segment as follows: For $i = 1, ..., n$, if $\tilde{b}_i \leqslant \tau$, then the target array of the (single) segment of A_i is of size $\min\{b_i', 2\tau\}$ (as argued in Section 8, this quantity is readily available). If $\tilde{b}_i > \tau$, on the other hand, the target array of each segment of A_i is of size 2τ, and the target arrays of all segments of A_i form a contiguous block of memory cells—this is easy to ensure, since they are all of the same size. Note that the total size of the target arrays is $O(n)$, so that they can be allocated according to Theorem 7.1 from a base array of size $O(n)$, which will be the output array of the semisorting.

We finally associate with each segment the task of placing $\min\{m, s\}$ elements stored in the segment in its target array, where m is the number of elements stored in the segment and s is the size of its target array, and execute all the tasks using operation allocation, as described in Section 7; if we take the length of a task to be the sum of the size of its associated segment and the size of the corresponding target array, the necessary prerequisites are easily seen to be satisfied (since every task is of length at most 3τ, it suffices to show how to process a task in linear sequential time, which is straightforward).

We want to show that with high probability, the number of elements not placed in the corresponding target arrays in the computation above is $O(n/\tau)$. To this end note that the choice of a list number in the scattering in time implicitly is a choice of a segment, and that the elements stored in a segment can be placed in the corresponding target array if their number is no larger than the size of the target array, i.e., if their number is at most 2τ. The expected number of elements of a well-estimated color class $\mathscr{B}_i$ choosing a particular segment is at most τ (since $b_i \leqslant \tilde{b}_i$, the number of lists associated with $\mathscr{B}_i$ at least equals the number of elements in $\mathscr{B}_i$). Hence by Chernoff bound (a), the probability that a fixed element of a well-estimated color class $\mathscr{B}_i$ finds itself in a segment containing $2\tau + 1$ or more elements of $\mathscr{B}_i$ is at most $e^{-\tau/3}$. A martingale argument now shows that with high probability, the number of noncolliding good elements that cannot be placed in the appropriate target arrays is $O(n/\tau)$. ∎

10. APPLICATIONS OF SEMISORTING

This section describes a few relatively straightforward applications of Theorem 9.14. A number of less immediate applications were mentioned in the introduction.

Our first goal is to extend the semisorting result to strong semisorting. Recall that whereas usual semisorting places the elements of each color class in a subarray of a base array, strong semisorting additionally requires the size of the subarray of each color class to be proportional to the size of the color class, a property that is often useful in applications.

Going from usual semisorting to strong semisorting obviously is a matter of compacting each color class into linear space. Treating color classes independently, we can use the algorithm of Theorem 5.3 to choose a suitable size for the destination array of each color and carry out the actual compaction using the algorithms of Section 4. Since color classes may be small, however, their sizes may be overestimated (as well as underestimated) by the algorithm of Theorem 5.3; as a result, although the compaction of a color class succeeds, it may fail in the sense that the destination array is too large. Since this is an infrequent event, we have enough resources to retry each unsuccessful compaction many times, which achieves a high reliability. An indispensable prerequisite for this, however, is the ability to tell whether a particular compaction was indeed into linear space. We therefore need a *certified approximate counting* algorithm that with high probability estimates the number of ones among n bits correctly, up to a constant factor, and that explicitly reports failure if it is unable to do so, i.e., a Las Vegas algorithm for approximate counting (Lemma 10.3). Our idea for obtaining such an algorithm is simple: Compacting the ones in the input into an array A furnishes a proof that their number b is at most $|A|$. On the other hand, subsequently compacting the free cells in A into an array Q proves that $|A| - b \leqslant |Q|$, which yields a lower bound on b.

We already know how to compact b elements into an array of size cb, where c is a constant. It turns out, however, that for the scheme above to work we cannot allow c to be arbitrarily large; in fact, we must demand that $c < 2$. We therefore briefly depart from our usual philosophy of ignoring constant factors to show that the relevant result in Section 4 (Theorem 4.6) actually holds for any constant $c > 1$ (Lemma 10.1). Observations similar to Lemmas 10.1 and 10.2 were made independently and first reported by Goodrich (1991).

LEMMA 10.1. *For every fixed $\mu > 0$ there is a constant $\varepsilon > 0$ such that for all given n, d, $\tau \in \mathbb{N}$ with $\tau \geqslant \log^* n$, complete compaction problems of size n and with parameters $d \rightarrow_{(1+\mu)d} 0$ can be solved on a* TOLERANT PRAM *using $O(\tau)$ time, $\lceil n/\tau \rceil$ processors and $O(n)$ space with probability at least $1 - 2^{-n^{\varepsilon}}$ (Las Vegas).*

Proof. Without loss of generality we can assume that $\mu d \geqslant 12$, since otherwise the number of active input elements is bounded by a constant, that $d \leqslant n$ and that μ is rational and at most 1. It suffices to describe a basic algorithm with a failure probability of $2^{-d^{\Omega(1)}}$, since for $d \leqslant \sqrt{n}$ the active elements can be compacted into an array of size $O(\sqrt{n})$ using the algorithm of Corollary 4.3, after which the basic algorithm can be applied independently $\Theta(\sqrt{n})$ times. It also suffices, for a certain constant $K \in \mathbb{N}$, to place all except at most $\mu d/K$ elements in an array of size $s = \lceil (1 + \mu/2)\, d \rceil$, since, provided that K is sufficiently large, the algorithm of Theorem 4.6 can then be used to place the remaining elements in an array of size $\lceil \mu d/3 \rceil$, which for $\mu d \geqslant 12$ is at most $(1 + \mu)\, d - s$.

We do this using repeated 1-scattering over a fixed array A of size s. Initially let all elements be active, and then carry out a number of stages. In each stage the remaining active elements are 1-scattered over A; colliding elements as well as elements that hit an element placed in a previous stage remain active, while the other elements are placed in A and become inactive.

Assume that some stage starts with more than $\mu d/K$ active elements. It is easy to see that a fixed element collides or hits an element placed in a previous stage with probability at most $d/s \leqslant 1/(1 + \mu/2)$, so that the expected number of elements deactivated in the stage is at least $(\mu d/K)(1 - 1/(1 + \mu/2)) = \mu^2 d/(2K(1 + \mu/2)) \geqslant (\mu^2/(4K)) \cdot d$. By Lemma 2.2(b), with high probability the stage under consideration deactivates at least $(\mu^2/(8K)) \cdot d$ elements. We may conclude that with high probability, $\lceil 8K/\mu^2 \rceil$ stages suffice to reduce the number of active elements to at most $\mu d/K$, as desired. ∎

LEMMA 10.2. *For every fixed $\mu > 0$ there is a constant $\varepsilon > 0$ such that for all given n, $\tau \in \mathbb{N}$ with $\tau \geqslant \log^* n$, the following problem can be solved on a* TOLERANT PRAM *using $O(\tau)$ time, $\lceil n/\tau \rceil$ processors and $O(n)$ space with probability at least $1 - 2^{-n^{\varepsilon}}$ (Monte Carlo): Given n bits $x_1, \ldots, x_n$, compute a nonnegative integer $\hat{b}$ such that $b \leqslant \hat{b} \leqslant (1 + \mu)\, b$, where $b = |\{j: 1 \leqslant j \leqslant n \text{ and } x_j = 1\}|$.*

Proof. Define an input element to be *active* if its value is 1, and assume without loss of generality that μ is rational and at most 1. Take $\theta = \mu/3$ and begin by using the algorithm of Theorem 5.3 to compute an integer $\tilde{b}$ such that with high probability, $b/K \leqslant \tilde{b} \leqslant b$, for some constant $K \geqslant 1$. If $\theta\tilde{b} \leqslant 1$, solve the problem in a trivial manner. Otherwise repeatedly use the algorithm of Lemma 10.1 with $\mu = \theta$ to attempt to compact the active elements with limit $d = \tilde{b}$, $\tilde{b} + \lfloor \theta\tilde{b} \rfloor$, $\tilde{b} + 2\lfloor \theta\tilde{b} \rfloor$, ..., stopping after the first successful complete compaction, and return as $\hat{b}$ the quantity $\lfloor (1 + \theta)\, d \rfloor$, where d is the limit of the last (successful) attempt.

The size of the destination array of the successful compaction is at most $(1 + \theta)\, d$; i.e., the relation $b \leqslant \hat{b}$ is satisfied. On the other hand, the compaction will succeed with high probability for any limit which is at least b. Provided that indeed $\tilde{b} \leqslant b$, the first limit with this property in the series above is at most $b + \theta\tilde{b} \leqslant (1 + \theta)\, b$, so that with high probability, $\hat{b} \leqslant (1 + \theta)^2\, b \leqslant (1 + 3\theta)\, b = (1 + \mu)\, b$. It is easy to see from this that provided that indeed $\tilde{b} = \Omega(b)$, with high probability the algorithm of Lemma 10.1 is applied only a constant number of times, i.e., the running time is $O(\tau)$. ∎

Informally, the "true" output of the Las Vegas algorithm below for approximate counting is the integer $\hat{b}$. The condition $y = 0$ indicates the correctness of the output, whereas $y = 1$ signifies that the execution failed.

LEMMA 10.3. *For every fixed $\mu > 0$ there is a constant $\varepsilon > 0$ such that for all given n, $\tau \in \mathbb{N}$ with $\tau \geqslant \log^* n$, the following problem can be solved on a* TOLERANT PRAM *using $O(\tau)$ time, $\lceil n/\tau \rceil$ processors and $O(n)$ space: Given n bits $x_1, \ldots, x_n$, compute a nonnegative integer $\hat{b}$ and a bit y such that*

(1) *If $y = 0$, then $b \leqslant \hat{b} \leqslant (1 + \mu)\, b$, where $b = \sum_{j=1}^{n} x_j$;*

(2) $\Pr(y = 1) \leqslant 2^{-n^{\varepsilon}}$.

Proof. Assume that μ is rational, choose $\theta < 1$ to make $(1 - 4\theta)(1 + \mu) = 1$ and begin by applying the algorithm of Lemma 10.2 to obtain a nonnegative integer $\tilde{b}$ such that with high probability, $b \leqslant \tilde{b} \leqslant b/(1 - \theta)$. Taking $\hat{b} = \lfloor (1 + \theta)\, \tilde{b} \rfloor$, we now verify the two inequalities $b \leqslant \hat{b} \leqslant (1 + \mu)\, b$ and set $y = 1$ if the verification fails. Assume that $\tilde{b} \geqslant 1$, since for $\tilde{b} = 0$ the verification can be done trivially according to Lemma 2.7.

Again define an input element to be active if its value is 1. Let A be an array of size $\hat{b}$ and use the algorithm of Lemma 10.1 with $\mu = \theta$ and $d = \tilde{b}$ to attempt to place the active elements in A. If this succeeds, it clearly proves that $b \leqslant \hat{b}$. On the other hand, since $\tilde{b} \geqslant b$ with high probability, the compaction succeeds with high probability.

Assuming that the compaction into A succeeds, we next use the algorithm of Lemma 10.1 with $\mu = 1$ and $d = \lfloor 2\theta\hat{b} \rfloor$ to attempt to place the free cells in A in an array Q of size $\lfloor 4\theta\hat{b} \rfloor$. More precisely, this entails deriving from A a bit sequence $x'_1, \ldots, x'_{|A|+n}$ such that $x'_j = 1$ if and only if the jth cell of A contains no input element, for $j = 1, \ldots, |A|$, and $x'_j = 0$ for $j = |A| + 1, \ldots, |A| + n$, and then using $x'_1, \ldots, x'_{|A|+n}$ as input to the algorithm of Lemma 10.1 ($x'_{|A|+1}, \ldots, x'_{|A|+n}$ are added only to ensure that the algorithm works correctly with high probability). Take $y = 0$ if and only if both compactions according to Lemma 10.1 succeed.

The following happens with high probability: $b \geqslant (1-\theta)\tilde{b}$, so the number $\hat{b}-b$ of free cells in A is at most $(1+\theta)\tilde{b}-(1-\theta)\tilde{b}=2\theta\tilde{b} \leqslant 2\theta\hat{b}$, and the compaction into Q succeeds. If it does, this is proof that the number of free cells in A is at most $|Q|$, and hence that $b \geqslant \hat{b}-\lfloor 4\theta\hat{b} \rfloor \geqslant (1-4\theta)\hat{b}$, from which it follows that $\hat{b} \leqslant (1+\mu)b$. ∎

THEOREM 10.4. *There is a constant $\varepsilon > 0$ such that for all given n, $\tau \in \mathbb{N}$ with $\tau \geqslant \log^* n$, n-color strong semisorting problems of size n can be solved on a* TOLERANT PRAM *using $O(\tau)$ time, $\lceil n/\tau \rceil$ processors and $O(n)$ space with probability at least $1-2^{-n^{\varepsilon}}$ (Las Vegas).*

Proof. We begin by semisorting the input according to Theorem 9.14. As in the proof of Corollary 9.12, we can view this as providing us with n disjoint subarrays $A_1, \ldots, A_n$ of a base array A of size $O(n)$ such that the elements of $\mathscr{B}_i$ are placed in A_i, for $i=1, \ldots, n$. Our goal is to move the elements in $\mathscr{B}_i$ from A_i to a subarray of A_i of size $O(b_i)$, for $i=1, \ldots, n$, which provides a solution to the strong semisorting problem.

We process $A_1, \ldots, A_n$ using operation allocation, as described in Section 7. The sequential processing of an array is simply exact compaction by means of prefix summation. The parallel processing of A_i is as follows, for $i=1, \ldots, n$: Apply the algorithm of Lemma 10.3 to A_i with $\mu=1$ to obtain a pair $(\hat{b}_i, y_i)$, where $\hat{b}_i$ is an estimate of b_i and y_i is an indication of the validity of $\hat{b}_i$ (if $y_i=0$, then $b_i \leqslant \hat{b}_i \leqslant 2b_i$). Subsequently apply the algorithm of Lemma 10.1 with $\mu=1$ to attempt to place $\mathscr{B}_i$ in a subarray A_i' of A_i of size at most $2\hat{b}_i$ (if $2\hat{b}_i \geqslant |A_i|$, simply take $A_i'=A_i$). If either $y_i=1$ or the compaction of $\mathscr{B}_i$ into A_i' fails, we will say that the processing of A_i fails. Take $y_i'=1$ if the processing of A_i fails, and $y_i'=0$ otherwise.

By Lemmas 10.1 and 10.3, the processing of A_i fails with probability at most $2 \cdot 2^{-|A_i|^{\delta}}$, for some fixed $\delta > 0$ and for $i=1, \ldots, n$. In particular, with high probability the processing of an array of size $n^{1/8}$ or more does not fail. As another consequence, $E(\sum_{i=1}^n y_i' \cdot 2^{|A_i|^{\delta}}) = O(n)$. Furthermore, by a martingale argument, $\sum_{i=1}^n y_i' q_i = O(n)$ with high probability, where $q_i = \min\{2^{\lceil |A_i|^{\delta} \rceil}, \lceil n^{1/4} \rceil\}$, for $i=1, \ldots, n$. However, this means that if the processing of A_i fails, for some $i \in \{1, \ldots, n\}$, then we can expend $\Theta(q_i)$ operations in a second attempt to process A_i. We again use operation allocation, now with a new collection of tasks. Since $|A_i| \leqslant n^{1/8}$ with high probability, we can clearly compact $\mathscr{B}_i$ exactly in $O(q_i)$ sequential time. Furthermore, if $q_i = 2^{\lceil |A_i|^{\delta} \rceil}$, we can use prefix summation (Lemma 2.4) to compact $\mathscr{B}_i$ in $O(\tau)$ time using $\lceil q_i/\tau \rceil$ processors, while if instead $q_i = \lceil n^{1/4} \rceil$, $\lceil q_i/\tau \rceil$ processors suffice to carry out $\Theta(n^{1/8})$ independent attempts to process A_i in $O(\tau)$ time as above, at least one of which will succeed with high probability. ∎

COROLLARY 10.5. *There is a constant $\varepsilon > 0$ such that for all given n, $\tau \in \mathbb{N}$ with $\tau \geqslant \log^* n$, n-color fine-profiling problems of size n can be solved on a* TOLERANT PRAM *using $O(\tau)$ time, $\lceil n/\tau \rceil$ processors and $O(n)$ space with probability at least $1-2^{-n^{\varepsilon}}$ (Las Vegas).*

Proof. Immediate from Theorem 10.4 and Lemma 2.8(a) (see the proof of Corollary 9.12). ∎

A second application of semisorting is to integer chain-sorting. Recall that the chain-sorting problem is to store given keys in sorted order in a linked list. In the formal definition below, the linked list is represented by a circular structure and a pointer to the last list element.

DEFINITION. For all $n, m \in \mathbb{N}$, the *m-color chain-sorting* problem of size n is the following: Given n integers $x_1, \ldots, x_n$ in the range $1..m$, compute a cyclic permutation $\pi_1, \ldots, \pi_n$ of $1, \ldots, n$ and an integer $q \in \{1, \ldots, n\}$ such that for all $j \in \{1, \ldots, n\} \setminus \{q\}$, we have $x_{\pi_j} \geqslant x_j$.

THEOREM 10.6. *There is a constant $\varepsilon > 0$ such that for all given n, $\tau \in \mathbb{N}$ with $\tau \geqslant \log^* n$, n-color chain-sorting problems of size n can be solved on a* TOLERANT PRAM *using $O(\tau)$ time, $\lceil n/\tau \rceil$ processors and $O(n)$ space with probability at least $1-2^{-n^{\varepsilon}}$ (Las Vegas).*

Proof. Begin by semisorting the input elements into an array A of size $O(n)$ according to Theorem 9.14. Using the algorithm of Lemma 2.8(a), it is then easy to construct a linked list containing precisely the elements of $\mathscr{B}_i$ in the order in which they occur in A, for $i=1, \ldots, n$. The remaining problem is to concatenate these lists in the right order. This can be done by applying the algorithm of Lemma 2.8(a) a second time, now to an n-bit input whose ith bit is 1 if and only if $\mathscr{B}_i \neq \varnothing$, for $i=1, \ldots, n$. ∎

A claim similar to Theorem 10.6 above was made previously in (Gil *et al.*, 1991). It seems unlikely, however, that any algorithm based on the outline given in (Gil *et al.*, 1991) can be made to run in linear space.

An important application of Theorem 10.6 is to (standard) integer sorting. Let us restrict attention to the problem of sorting n integers in the range $1..n$ on a CRCW PRAM. Rajasekaran and Reif (1989) describe a randomized algorithm with optimal speedup for this problem that uses $O(\log n)$ time and $O(n/\log n)$ processors with high probability. Bhatt *et al.* (1991) give a deterministic algorithm that works in $O(\log n/\log\log n)$ time using $O(n(\log\log n)^2/\log n)$ processors. We show how to combine the time bound of (Bhatt *et al.*, 1991) with the time-processor product of (Rajasekaran and Reif, 1989), thus achieving at the same time optimal speed and optimal speedup. Similar results were found independently by Matias and Vishkin (1991) and Raman (1991); note, however, that the algorithms of these authors (which are quite similar) are inherently much less reliable than the algorithm given here—the failure probability is $\Omega(2^{-(\log n)^{\alpha}})$, for some fixed α, to be contrasted with our failure probability of $2^{-n^{\Omega(1)}}$.

Our algorithm makes use of a subroutine for monotonic list ranking with optimal speedup. The *monotonic list ranking* problem of size n is, given a linked list of n elements such that which of two given list elements precedes the other can be determined in constant time by a single processor, to mark each element of the list with its position within the list.

LEMMA 10.7 (Bhatt *et al.*, 1991). *For all given integers $n \geqslant 4$ and $\tau \geqslant \log n/\log\log n$, monotonic list ranking problems of size n can be solved on a (deterministic)* TOLERANT PRAM *using $O(\tau)$ time, $\lceil n/\tau \rceil$ processors and $O(n)$ space.*

THEOREM 10.8. *There is a constant $\varepsilon > 0$ such that for all given integers $n \geqslant 4$ and $\tau \geqslant \log n/\log\log n$, n integers in the range $1..n$ can be sorted on a* TOLERANT PRAM *using $O(\tau)$ time, $\lceil n/\tau \rceil$ processors and $O(n)$ space with probability at least $1 - 2^{-n^{\varepsilon}}$ (Las Vegas).*

Proof. Chain-sort the input elements using the algorithm of Theorem 10.6 and compute the position of each element within the resulting list using the algorithm of Lemma 10.7. In order to determine the relative order of two elements with the same value, compare their positions in the semisorted array output by the algorithm of Theorem 9.14. ∎

11. NONOPTIMAL ALGORITHMS

This section investigates the effect for the problems considered of allowing slightly superlinear processor and space bounds. In some cases, we also have to generalize the problems by introducing a so-called *slack parameter* (this notion already appeared in Lemmas 6.1 and 7.3). We begin by showing that compaction with slack can be done in constant time.

Although, technically, the results stated in this section allow k and τ to vary independently as functions of n, it is probably most useful to imagine that $\tau = k$ is constant. Our informal discussion makes this assumption.

THEOREM 11.1. *There is a constant $\varepsilon > 0$ such that for all given n, d, k, $\tau \in \mathbb{N}$ with $\tau \geqslant k$, complete compaction problems of size n and with parameters $d \rightarrow_{O(s)} 0$, where $s = d\lfloor \log^{(k)} n \rfloor$, can be solved on a* TOLERANT PRAM *using $O(\tau)$ time, $\lceil kn/\tau \rceil$ processors and $O(n)$ space with probability at least $1 - 2^{-n^{\varepsilon}}$ (Las Vegas).*

Proof. We can assume that $(2\lceil \log^{(k+1)} d \rceil)^3 \leqslant \log^{(k)} d$, since otherwise $k = \Omega(\log^* d)$ and we can apply the algorithm of Theorem 4.6. Then apply the algorithm of Lemma 4.4 $O(k)$ times. The number of operations needed is $O(kn)$, which translates into $\lceil kn/\tau \rceil$ processors, for any $\tau \geqslant k$. Omitting the size σ of the destination array from the notation $d_1 \rightarrow_{\sigma} d_2$, we can express the process symbolically as follows:

$$d \leqslant \frac{d\lfloor \log^{(k)} d \rfloor}{(2\lceil \log^{(k+1)} d \rceil)^3} \rightarrow \frac{d\lfloor \log^{(k)} d \rfloor}{2^{6\lceil \log^{(k+1)} d \rceil}}$$

$$\leqslant \frac{d}{\lceil \log^{(k)} d \rceil^3} \rightarrow \frac{d}{\lceil \log^{(k-1)} d \rceil^3} \rightarrow \cdots \rightarrow \frac{d}{\lceil \log d \rceil^3} \rightarrow \frac{d}{d^3}.$$

The last step in this sequence reduces the number of active elements below 1, i.e., to zero. The destination array used in the first step is of size $O(d \log^{(k)} d) = O(s)$, and the sizes of the destination arrays used in the remaining steps sum to $O(d)$. Hence all active elements can indeed be placed in an array of size $O(s)$. ∎

We now extend Theorem 11.1 to the case of several colors. In contrast with the algorithm of Theorem 11.1, the generalized algorithm of Theorem 11.2 needs superlinear space.

DEFINITION. For all n, $m \in \mathbb{N}$, $d_1, \ldots, d_m \geqslant 0$ and $\lambda \geqslant 1$, the complete colored compaction problem of size n and with limits $d_1, \ldots, d_m$ and slack λ is, given n integers $x_1, \ldots, x_n$ in the range $0..m$ such that $|\{j: 1 \leqslant j \leqslant n \text{ and } x_j = i\}| \leqslant d_i$, for $i = 1, \ldots, m$, to compute a complete placement for $x_1, \ldots, x_n$ with bounds $\lambda d_1, \ldots, \lambda d_m$.

THEOREM 11.2. *There is a constant $\varepsilon > 0$ such that for all given n, m, k, τ, $d_1, \ldots, d_m \in \mathbb{N}$ with $m = (\log n)^{O(1)}$ and $\tau \geqslant k$, complete colored compaction problems of size n with limits $d_1, \ldots, d_m$ and with slack $O(\log^{(k)} n)$ can be solved on a* TOLERANT PRAM *using $O(\tau)$ time, $\lceil kn/\tau \rceil$ processors and $O(n + \sum_{i=1}^{m} d_i \log^{(k)} n)$ space with probability at least $1 - 2^{-n^{\varepsilon}}$ (Las Vegas).*

Proof. The idea of the proof is to apply the nonoptimal part of the algorithm of Theorem 6.2 (with $\delta = 1/2$) in a situation in which some of the elements have already been deactivated. For $i = 1, \ldots, m$, assume that $d_i \leqslant n$ and define $\mathscr{B}_i$ to be *large* if $b_i > n^{1/4}$. Recall that the algorithm of Theorem 6.2 essentially applies the algorithm of Lemma 6.1 $\log^* n$ times to reduce the fraction of active elements in each large color class below a certain threshold, after which the compaction is finished using negligible resources. In the present setting, where we are allowed $O(\log^{(k)} n)$ slack, we can speed up the deactivation by first 1-scattering the elements of $\mathscr{B}_i$ over an array of size $2d_i\lceil \log^{(k)} n \rceil$, for $i = 1, \ldots, m$. Lemma 3.6(b) shows that the number of elements in a fixed large color class $\mathscr{B}_i$ that collide in the 1-scattering is at most $b_i/\log^{(k)} n$, except with probability at most $e^{-\zeta}$, where $\zeta = (b_i/\log^{(k)} n)^2/(32b_i) = \Omega(b_i/(\log^{(k)} n)^2) = \Omega(n^{1/8})$, so that with high probability the fraction of active elements left in any large color class is at most $1/\log^{(k)} n$. It is now easy to see that all but the last $O(k)$ applications of the algorithm of Lemma 6.1 in the algorithm of Theorem 6.2 can be omitted. Since all subroutines used can be made to run in $O(\tau/k)$ time using $\lceil kn/\tau \rceil$ processors, we can therefore deactivate all elements in $O(\tau)$ time. ∎

Armed with Theorem 11.2, we can easily use the reductions of interval allocation and interval marking to colored compaction given in Section 7 to derive similar results for these problems.

DEFINITION. For all $n \in \mathbb{N}$ and $\lambda \geqslant 1$, the interval allocation problem of size n and with slack λ is, given n nonnegative integers $x_1, ..., x_n$, to compute a complete interval placement for $x_1, ..., x_n$ with slack λ.

THEOREM 11.3. *There is a constant $\varepsilon > 0$ such that for all given n, k, $\tau \in \mathbb{N}$ with $\tau \geqslant k$, interval allocation problems of size n and with slack $O(\log^{(k)} n)$ can be solved on a* TOLERANT PRAM *using $O(\tau)$ time, $\lceil kn/\tau \rceil$ processors and $O(n \log^{(k)} n)$ space with probability at least $1 - 2^{-n^\varepsilon}$ (Monte Carlo).*

Proof. As the proof of Theorem 7.1, except that Theorem 11.2 is used instead of Theorem 6.2. ∎

In (Bast *et al.*, 1992), Theorem 11.3 is used to prove a related result: For any $k \in \mathbb{N}$, usual interval allocation problems (i.e., with constant slack) of size n can be solved in $O(k)$ time using $O(n \log^{(k)} n)$ processors and $O(n \log^{(k)} n)$ space with high probability.

DEFINITION. For all $n \in \mathbb{N}$ and $\lambda \geqslant 1$, the interval marking problem of size n and with slack λ is the following: Given n nonnegative integers $x_1, ..., x_n$, compute nonnegative integers $s, z_1, ..., z_s$ such that

(1) For all integers i, j, k with $1 \leqslant i \leqslant j \leqslant k \leqslant s$, if $z_i = z_k \neq 0$, then $z_j = z_i$;

(2) For $i = 1, ..., n$, $|\{j: 1 \leqslant j \leqslant s \text{ and } z_j = i\}| = x_i$;

(3) $s = O(\lambda \sum_{j=1}^{n} x_j)$.

THEOREM 11.4. *There is a constant $\varepsilon > 0$ such that for all given n, k, $\tau \in \mathbb{N}$ with $\tau \geqslant k$, interval marking problems of size n and with slack $O(\log^{(k)} n)$ can be solved on a* TOLERANT PRAM *using $O(\tau)$ time, $\lceil (kn + W \log^{(k)} n)/\tau \rceil$ processors and $O((n + W) \log^{(k)} n)$ space with probability at least $1 - 2^{-n^\varepsilon}$ (Monte Carlo), where W is the sum of the input numbers.*

Proof. As the proof of Theorem 7.2, using Theorem 11.3 instead of Theorem 7.1. ∎

We next turn to the coarse-profiling problem, for which there is no need to introduce a slack parameter.

THEOREM 11.5. *There is a constant $\varepsilon > 0$ such that for all given $n, k \in \mathbb{N}$, n-color coarse-profiling problems of size n can be solved on a* TOLERANT PRAM *using $O(k)$ time, $O(n \log^{(k)} n)$ processors and $O(n \log^{(k)} n)$ space with probability at least $1 - 2^{-n^\varepsilon}$ (Monte Carlo).*

Proof. If the allocation of space and processors in the algorithm of Lemma 8.3 is carried out using the algorithms of Theorems 11.3 and 11.4, the only part of the algorithm that needs more than constant time is the computation of $v_1, ..., v_n$ in lines (6)–(7). We will show how to compute $v_1, ..., v_n$ in constant time using $O(n \log^{(k)} n)$ processors and $O(n \log^{(k)} n)$ space. Fix $i \in \{1, ..., n\}$ and recall that in the algorithm of Lemma 8.3, the variable v_i steps through a sequence of values until hitting either r or the index of a nonfull row in a GCS $\mathscr{S}_i$, which becomes its final value.

Let $T = \lfloor \log^{(k+2)} n \rfloor$ and take $w_1 = 1$ and $w_t = \min\{2^{w_{t-1}}, T\}$, for $t = 2, ..., T$. Except for a suffix of length bounded by $k + O(1)$, the sequence of successive values assumed by v_i is a prefix of $w_1, ..., w_T$. It is easy to see from this that it suffices to determine the first nonfull row of $\mathscr{S}_i$, if any, among rows $w_1, ..., w_T$.

Given $w_1, ..., w_T$, it is a trivial matter to construct a table that maps w_t to t, for all $t \in \{1, ..., T\}$ with $w_t < T$, and that maps all other integers in the range $1..n$ to zero. Such a table allows each processor associated with an element participating in $\mathscr{S}_i$ to learn whether it occupies a position in a row of the form w_t, for some $t \in \{1, ..., T\}$ with $w_t < T$, and, if so, for which value of t. We can now appeal to Lemma 3.4(a), which shows that the first nonfull row of $\mathscr{S}_i$ among rows $w_1, ..., w_T$, if any, can be found using $O(T)$ additional processors and $O(T)$ space, as desired.

We still need to describe how to obtain $w_1, ..., w_T$ in constant time. Let Δ be the set of sequences of length T with elements drawn from $\{1, ..., T\}$. Since $|\Delta| = T^T = O(\log^{(k)} n)$, we can associate a *team* of T processors with each element of Δ without using more than $\Theta(n)$ processors. Now let each team decide in constant time whether its associated sequence is $w_1, ..., w_T$, simply by verifying the T local conditions $w_1 = 1$ and $w_t = \min\{2^{w_{t-1}}, T\}$, for $t = 2, ..., T$, and, if so, output its associated sequence. ∎

Theorem 11.5 represents the best that we can do on the TOLERANT PRAM; in particular, the number of processors needed is superlinear. On the ARBITRARY PRAM, on the other hand, we obtain a constant-time algorithm with optimal speedup with the sole drawback of superlinear space requirements.

THEOREM 11.6. *There is a constant $\varepsilon > 0$ such that for all given $n, k \in \mathbb{N}$, n-color coarse-profiling problems of size n can be solved on an* ARBITRARY PRAM *using $O(k)$ time, n processors and $O(n \log^{(k)} n)$ space with probability at least $1 - 2^{-n^\varepsilon}$ (Monte Carlo).*

Proof. Consider the algorithm of Theorem 11.5 and note that the processors allocated by the algorithm serve exclusively to evaluate (some of) the rows of a number of graduated conditional scatterings according to Lemma 3.4(a). By Lemma 3.4(b), the same computation can be carried out on the ARBITRARY PRAM with just one processor per GCS in addition to those associated with the elements participating in the GCS, so that clearly n processors suffice. Finally observe that the allocation of space can be done using the algorithm of Theorem 11.3. ∎

We finally derive a constant-time algorithm for a variant of semisorting defined below. Recall from Section 9 that the n-processor semisorting algorithm proceeds in a number of stages, each except the last of which essentially performs bootstrapping for the following stage, while only the last stage actually solves the entire problem. The key observation for the present section is that if we simply omit the bootstrapping of the first stages, the algorithm still operates in a well-defined way, but with a certain increase in its resource requirements and, possibly, a certain degradation in the quality of its output. It even turns out that if we start with Stage t_0, for some t_0, the bootstrapping effect of Stage t_0 is not affected by the absence of Stages $1, ..., t_0 - 1$, so that our original analysis applies without modification to all stages following Stage t_0. It therefore suffices to reanalyze Stage t_0 with respect to its resource requirements and its effect on the output.

DEFINITION. For all n, $m \in \mathbb{N}$ and $\lambda \geqslant 1$, the *m-color semisorting* problem of size n and with slack λ is the following: Given n integers $x_1, ..., x_n$ in the range $0..m$, compute n nonnegative integers $y_1, ..., y_n$ such that

(1) For $1 \leqslant i < j \leqslant n$, if $x_i \neq 0$, then $y_i \neq y_j$;

(2) For all $i, j, k \in \{1, ..., n\}$, if $y_i < y_j < y_k$ and $x_i = x_k$, then $x_j = x_i$;

(3) $\max\{y_j: 1 \leqslant j \leqslant n\} = O(\lambda n)$.

THEOREM 11.7. *There is a constant $\varepsilon > 0$ such that for all given n, $k \in \mathbb{N}$, n-color semisorting problems of size n and with slack $O(\log^{(k)} n)$ can be solved on a* TOLERANT PRAM *using $O(k)$ time, $O(n \log^{(k)} n)$ processors and $O(n \log^{(k)} n)$ space with probability at least $1 - 2^{-n^{\varepsilon}}$ (Las Vegas).*

Proof. Begin by computing a coarse-profile for the input numbers using the algorithm of Theorem 11.5. Then execute only Stages $t_0, ..., T$ of the algorithm of Lemma 9.1, where $t_0 = \max\{1, T - 2(k+1)\}$; note that $v_{t_0} = O(\log^{(k+1)} n)$. Recall that each stage deactivates elements by placing them in suitably-sized arrays, one for each color. In every stage, the size of the array used for a particular color is chosen on the basis of a test scattering for that color, which roughly estimates the number of remaining active elements of that color. As an important consequence of this "self-correcting" mechanism, we were able to analyze the deactivation capability of a stage without relying on the deactivation carried out in earlier stages (if earlier stages perform poorly, the resource requirements of the stage at hand go up, but it will still reduce the number of remaining active elements to the required level). Therefore Lemma 9.5 remains true (the last stage always deactivates all remaining active elements) and, with the additional restriction $t \geqslant t_0$, the same holds for Lemmas 9.6–9.9. We must show that the algorithm is correct and bound its resource requirements. As in Section 9, this essentially boils down to bounding the total size of the arrays $A_{i,t}$ allocated in Stage t, for $t = t_0, ..., T$. For $t \geqslant t_0 + 1$, this quantity can be seen to be $O(n/v_t^3)$ (Lemma 9.10); the reason is that in the analysis of a particular stage, Lemmas 9.7–9.9 can be applied to the previous stage. As regards Stage t_0 itself, it is easy to see from lines (6), (8), (13), (14) and (16) in the algorithm that the total size of the arrays allocated in Stage t_0 is $O(nv_{t_0}^{19})$. Since $v_{t_0}^{22} = O(\log^{(k)} n)$, by the choice of t_0, it now follows essentially as in the proof of Lemma 9.11 that with high probability, the algorithm uses $O(n \log^{(k)} n)$ processors and $O(n \log^{(k)} n)$ space and solves the semisorting problem with slack $O(\log^{(k)} n)$. ∎

COROLLARY 11.8. *There is a constant $\varepsilon > 0$ such that for all given n, $k \in \mathbb{N}$, n-color chain-sorting problems of size n can be solved on a* TOLERANT PRAM *using $O(k)$ time, $O(n \log^{(k)} n)$ processors and $O(n \log^{(k)} n)$ space with probability at least $1 - 2^{-n^{\varepsilon}}$ (Las Vegas).*

Proof. As the proof of Theorem 10.6, using Theorem 11.7 instead of Theorem 9.14 and part (b) of Lemma 2.8 instead of part (a). ∎

ACKNOWLEDGMENTS

We are grateful to the anonymous referees for their meticulous reading and for many useful suggestions. We also thank Peter Miltersen for pointing us to the papers by Stockmeyer and Ajtai and Ben-Or, and Prabhakar Ragde for simplifying the proof of Lemma 3.5.

Received May 29, 1991; final manuscript received April 4, 1995

REFERENCES

Ajtai, M., and Ben-Or, M. (1984), A theorem on probabilistic constant depth computations, *in* "Proc. 16th Annual ACM Symposium on Theory of Computing," pp. 471–474.

Alon, N., and Megiddo, N. (1994), Parallel linear programming in fixed dimension almost surely in constant time, *J. Assoc. Comput. Mach.* **41**, 422–434.

Bast, H., Dietzfelbinger, M., and Hagerup, T. (1992), A perfect parallel dictionary, *in* "Proc. 17th International Symposium on Mathematical Foundations of Computer Science," Lecture Notes in Computer Science, Vol. 629, pp. 133–141, Springer-Verlag, Berlin/New York.

Bast, H., and Hagerup, T. (1991), Fast and reliable parallel hashing, *in* "Proc. 3rd Annual ACM Symposium on Parallel Algorithms and Architectures," pp. 50–61.

Beame, P., and Hastad, J. (1989), Optimal bounds for decision problems on the CRCW PRAM, *J. Assoc. Comput. Mach.* **36**, 643–670.

Berkman, O., and Vishkin, U. (1993), Recursive star-tree parallel data structure, *SIAM J. Comput.* **22**, 221–242.

Bhatt, P. C. P., Diks, K., Hagerup, T., Prasad, V. C., Radzik, T., and Saxena, S. (1991), Improved deterministic parallel integer sorting, *Inform. and Comput.* **94**, 29–47.

Bollobás, B. (1987), Martingales, isoperimetric inequalities and random graphs, *Colloq. Math. Soc. J. Bolyai* **52**, 113–139.

Chaudhuri, S., and Hagerup, T. (1994), Prefix graphs and their applications, *in* "Proc. 20th International Workshop on Graph-Theoretic Concepts in Computer Science," Lecture Notes in Computer Science, Vol. 903, pp. 206–218, Springer-Verlag, Berlin/New York.

Chlebus, B. S., Diks, K., Hagerup, T., and Radzik, T. (1988), Efficient simulations between concurrent-read concurrent-write PRAM models, *in* "Proc. 13th Symposium on Mathematical Foundations of Computer Science," Lecture Notes in Computer Science, Vol. 324, pp. 231–239, Springer-Verlag, Berlin/New York.

Chlebus, B. S., Diks, K., Hagerup, T., and Radzik, T. (1989), New simulations between CRCW PRAMs, *in* "Proc. 7th International Conference on Fundamentals of Computation Theory," Lecture Notes in Computer Science, Vol. 380, pp. 95–104, Springer-Verlag, Berlin/New York.

Cole, R., and Vishkin, U. (1989), Faster optimal parallel prefix sums and list ranking, *Inform. and Comput.* **81**, 334–352.

Fich, F. E., Ragde, P., and Wigderson, A. (1988a), Simulations among concurrent-write PRAMs, *Algorithmica* **3**, 43–51.

Fich, F. E., Ragde, P., and Wigderson, A. (1988b), Relations between concurrent-write models of parallel computation, *SIAM J. Comput.* **17**, 606–627.

Gil, J. (1990), "Lower Bounds and Algorithms for Hashing and Parallel Processing," Ph.D. Thesis, The Hebrew University, Jerusalem.

Gil, J. (1994), Renaming and dispersing: Techniques for fast load balancing, *J. Parallel Distrib. Comput.* **23**, 149–157.

Gil, J., Matias, Y., and Vishkin, U. (1991), Towards a theory of nearly constant time parallel algorithms, *in* "Proc. 32nd Annual Symposium on Foundations of Computer Science," pp. 698–710.

Goodrich, M. T. (1991), Using approximation algorithms to design parallel algorithms that may ignore processor allocation, *in* "Proc. 32nd Annual Symposium on Foundations of Computer Science," pp. 711–722.

Grolmusz, V. (1991), Large parallel machines can be extremely slow for small problems, *Algorithmica* **6**, 479–489.

Grolmusz, V., and Ragde, P. (1987), Incomparability in parallel computation, *in* "Proc. 28th Annual Symposium on Foundations of Computer Science," pp. 89–98; *Discrete Appl. Math.* **29** (1990), 63–78.

Hagerup, T. (1992a), Fast and optimal simulations between CRCW PRAMs, *in* "Proc. 9th Annual Symposium on Theoretical Aspects of Computer Science," Lecture Notes in Computer Science, Vol. 577, pp. 45–56, Springer-Verlag, Berlin/New York.

Hagerup, T. (1992b), The log-star revolution, *in* "Proc. 9th Annual Symposium on Theoretical Aspects of Computer Science," Lecture Notes in Computer Science, Vol. 577, pp. 259–278, Springer-Verlag, Berlin/New York.

Hagerup, T. (1992c), On a compaction theorem of Ragde, *Inform. Process. Lett.* **43**, 335–340.

Hagerup, T. (1995), The parallel complexity of integer prefix summation, *Inform. Process. Lett.* **56**, 59–64.

Hagerup, T., and Katajainen, J. (1993), Improved parallel bucketing algorithms for proximity problems, *in* "Proc. 26th Hawaii International Conference on System Sciences, Vol. II: Software Technology," pp. 318–327.

Hagerup, T., and Radzik, T. (1990), Every robust CRCW PRAM can efficiently simulate a PRIORITY PRAM, *in* "Proc. 2nd Annual ACM Symposium on Parallel Algorithms and Architectures," pp. 117–124.

Hagerup, T., and Raman, R. (1992), Waste makes haste: Tight bounds for loose parallel sorting, *in* "Proc. 33rd Annual Symposium on Foundations of Computer Science," pp. 628–637.

Hagerup, T., and Rüb, C. (1990), A guided tour of Chernoff bounds, *Inform. Process. Lett.* **33**, 305–308.

MacKenzie, P. D. (1992), Load balancing requires $\Omega(\log^* n)$ expected time, *in* "Proc. 3rd Annual ACM-SIAM Symposium on Discrete Algorithms," pp. 94–99.

Matias, Y., and Vishkin, U. (1991), Converting high probability into nearly-constant time—with applications to parallel hashing, *in* "Proc. 23rd Annual ACM Symposium on Theory of Computing," pp. 307–316.

McDiarmid, C. (1989), On the method of bounded differences, *in* "Surveys in Combinatorics" (J. Siemons, Ed.), London Math. Soc. Lecture Note Series 141, pp. 148–188, Cambridge Univ. Press, Cambridge, UK.

Ragde, P. (1993), The parallel simplicity of compaction and chaining, *J. Algorithms* **14**, 371–380.

Rajasekaran, S., and Reif, J. H. (1989), Optimal and sublogarithmic time randomized parallel sorting algorithms, *SIAM J. Comput.* **18**, 594–607.

Raman, R. (1990), The power of collision: Randomized parallel algorithms for chaining and integer sorting, *in* "Proc. 10th Conference on Foundations of Software Technology and Theoretical Computer Science," Lecture Notes in Computer Science, Vol. 472, pp. 161–175, Springer-Verlag, Berlin/New York.

Raman, R. (1991), "Optimal Sub-logarithmic Time Integer Sorting on the CRCW PRAM," Tech. Rep. no. 370, Dept. of Computer Science, Univ. of Rochester, Rochester, New York.

Reischuk, R. (1985), Probabilistic parallel algorithms for sorting and selection, *SIAM J. Comput.* **14**, 396–409.

Shiloach, Y., and Vishkin, U. (1982), An $O(\log n)$ parallel connectivity algorithm, *J. Algorithms* **3**, 57–67.

Stockmeyer, L. (1983), The complexity of approximate counting, *in* "Proc. 15th Annual ACM Symposium on Theory of Computing," pp. 118–126.

Valiant, L. G. (1990), General purpose parallel architectures, *in* "Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity" (J. van Leeuwen, Ed.), pp. 943–971, Elsevier, Amsterdam.