MAX-PLANCK-INSTITUT FÜR INFORMATIK

Waste Makes Haste: Tight Bounds for Loose Parallel Sorting

Torben Hagerup Rajeev Raman

MPI-I-92-141 September 1992

Im Stadtwald
66123 Saarbrücken
Germany

Waste Makes Haste: Tight Bounds for Loose Parallel Sorting

Torben Hagerup Rajeev Raman

MPI-I-92-141 September 1992

Waste Makes Haste: Tight Bounds for Loose Parallel Sorting*

TORBEN HAGERUP RAJEEV RAMAN
Max-Planck-Institut für Informatik, Im Stadtwald, W–6600 Saarbrücken, Germany
e-mail:{torben,raman}@mpi-sb.mpg.de

Abstract

Conventional parallel sorting requires the n input keys to be output in an array of size n, and is known to take $\Omega(\log n/\log\log n)$ time using any polynomial number of processors. The lower bound does not apply to the more "wasteful" convention of *padded sorting*, which requires the keys to be output in sorted order in an array of size $(1+o(1))n$. We give very fast randomized CRCW PRAM algorithms for several padded-sorting problems.

Applying only pairwise comparisons to the input and using kn processors, where $2 \leq k \leq n$, we can padded-sort n keys in $O(\log n/\log k)$ time with high probability (whp), which is the best possible (expected) run time for any comparison-based algorithm. We also show how to padded-sort n independent random numbers in $O(\log^* n)$ time whp with $O(n)$ work, which matches a recent lower bound, and how to padded-sort n integers in the range $1..n$ in constant time whp using n processors. If the integer sorting is required to be stable, we can still solve the problem in $O(\log\log n/\log k)$ time whp using kn processors, for any k with $2 \leq k \leq \log n$. The integer sorting results require the nonstandard OR PRAM; alternative implementations on standard PRAM variants run in $O(\log\log n)$ time whp. As an application of our padded-sorting algorithms, we can solve approximate prefix summation problems of size n with $O(n)$ work in constant time whp on the OR PRAM, and in $O(\log\log n)$ time whp on standard PRAM variants.

1 Introduction

Sorting is a fundamental problem that has been studied extensively in both parallel and sequential settings. The development of very fast parallel sorting algorithms has been hindered because it is sometimes harder to organize the output in the desired fashion than to actually compute the ordering of the input keys. For example, the usual output convention for sorting is that the sorted keys appear in consecutive memory locations. With this convention, sorting even a 0–1 input of size n on the powerful PRIORITY (CRCW) PRAM needs $\Omega(\log n/\log\log n)$ time using $n^{O(1)}$ processors [7].

With the recent spate of developments in the area of almost constant-time parallel algorithms (the so-called "log-star revolution" [21]) it is clear that near-logarithmic run times cannot automatically be deemed satisfactory any more. The potential of much faster algorithms is an excellent incentive for carefully reviewing the reasons for sorting the input and, when possible, to use alternative output conventions that are not subject to the lower bound. One such convention is that of *chain-sorting*, which requires the keys to be output in a sorted linked list.

*This work was supported by the ESPRIT II Basic Research Actions Program of the EC under contract No. 3075 (project ALCOM), and partly carried out while the first author was with the Departament de LSI of the Universitat Politècnica de Catalunya in Barcelona, Spain.

Hagerup [19, 20] demonstrated that small integers can be chain-sorted much faster than they can be sorted in the usual sense. Unfortunately, a major reason for sorting is to enable fast searching, and a chain-sorted output is inadequate for this purpose.

In this paper we consider *padded sorting*, which is much closer to the usual definition of sorting. Here the n input keys are required to be output in sorted order in an array of size at most $(1+\lambda)n$, for some $\lambda \geq 0$. λ is called the *padding factor* and should be $o(1)$ and, in general, as small as possible. The empty locations in a padded array can be filled quickly with copies of the nearest preceding key, so the output of a padded sorting suffices for searching. The lower bound of [7] does not hold for padded sorting, a fact that MacKenzie and Stout [26] and Sarnath [37] exploited to give faster padded-sorting algorithms. In the sequential context, padded arrays were proposed by Itai *et al.* [25] and Willard [39] as an alternative to normal arrays because insertions into padded arrays can be processed quickly.

We show that in a variety of parallel sorting problems, a padded output can be obtained dramatically faster than a normal one; in particular, the results of [26, 37] are improved. When nothing else is stated, our model of computation is the standard ARBITRARY (CRCW) PRAM; the results also hold for the weaker TOLERANT (CRCW) PRAM. Some results use the OR (CRCW) PRAM [6], where concurrent writing of several values to the same cell results in the bitwise OR of these values being stored in the cell. The word size is limited to $O(\log(n+p))$ bits, where n is the input size and p is the number of processors of the machine under consideration. As to the computational power of the OR PRAM, note on the one hand that an n-processor OR PRAM can simulate one step of an n-processor ARBITRARY PRAM in constant time whp [6], and on the other hand that with the above limit on the word size, a standard CRCW PRAM can simulate one step of an OR PRAM in constant time with a polynomial blowup in the number of processors, so that the lower bound of [7] still holds for the OR PRAM.

We now describe our results in more detail. When stating that the run time of a randomized algorithm is $\tilde{O}(E)$, where E is some expression, what we mean is that whp the run time is $O(E)$. All logarithms are to the base 2.

1.1 Comparison-Based Sorting

Here nothing may be assumed about the keys, except that any two of them can be compared in constant time. If the desired run time is logarithmic or slower, the problem has been satisfactorily solved: For every $p \leq n$, n keys can be sorted in $O(n \log n/p)$ time on a p-processor EREW PRAM [1, 12]. The problem that we consider is therefore to sort n keys using kn processors, where $k \geq 2$, in a comparison-based setting.

It is known that in the parallel comparison-tree (PCT) model, solving this problem requires $\Omega(\log n/\log k)$ (expected) time, for $2 \leq k \leq n$ [4, 3, 10]. Lower bounds for the PCT model apply to all purely comparison-based parallel algorithms. A matching upper bound was given in [4] *for the PCT model.* Upper bounds for the PCT model do not carry over to the PRAM.

On the PRAM, no algorithm is known that matches the PCT lower bound *and* has a run time of $o(\log n/\log\log n)$. Rajasekaran and Reif [33] gave an algorithm that runs in $\tilde{O}(\log n/\log\log n)$ time using $O(n(\log n)^\epsilon)$ processors, for arbitrary fixed $\epsilon > 0$, which matches the lower bound; their algorithm is easily modified to match the lower bound for *smaller* values of k as well. The best known deterministic solution to the problem uses $O(\log n/\log\log k)$ time for $4 \leq k \leq 2^n$ [12] — a super-polynomial number of processors is needed to achieve fast run times. Sarnath [37] gave a deterministic padded-sorting algorithm that takes $O(\log\log n)$ time on the average to padded-sort n keys using $O(n^{1+\epsilon})$ processors, for arbitrary fixed $\epsilon > 0$, under the assumption

that all permutations of the input keys are equally likely. This does not match the lower bound — with $\Theta(n^{1+\epsilon})$ processors the lower bound is $\Omega(1)$ — and the algorithm also has a padding factor of $\lambda = \Theta(n^{\epsilon})$, which may be too large for some applications. We show:

Theorem: *n keys can be padded-sorted using comparisons only in $\tilde{O}(\log n/\log k)$ time with kn processors, for any k with $2 \leq k \leq n$.*

This is the best possible run time for any comparison-based algorithm, regardless of the output convention; for example, the run time is $\tilde{O}(1)$ with $\Theta(n^{1+\epsilon})$ processors, for arbitrary fixed $\epsilon > 0$. We actually show a tradeoff between run time and padding factor: the more space we allow for the output, the faster we can sort ("more waste — more haste"). At one extreme of the tradeoff — for polylogarithmic k — the padding factor drops below $1/n$, hence to zero, and we get the result of Rajasekaran and Reif. A recent matching lower bound, due to Sarnath [38], shows our padded-sorting algorithm to be as fast as possible for the given padding factor. We provide a new and simpler proof of the lower bound.

1.2 Sorting Random Numbers

The problem of sorting n random numbers drawn independently from the uniform distribution over the unit interval $(0, 1]$ has a long history. It was first considered by McLaren [31], who gave a sequential algorithm with a linear expected run time (a survey of work on this and related problems can be found in [14]); parallel algorithms using $\tilde{O}(\log n)$ time and $O(n)$ operations were later developed [11, 18]. MacKenzie and Stout [26] showed that a random input distributed as described can be padded-sorted in $\tilde{O}(\log\log n)$ time with $O(n)$ work. This was subsequently improved to $\tilde{O}(\log\log n/\log\log\log n)$ time and $O(n)$ work by the original authors [27] and in [20]. MacKenzie [28] recently showed that any solution to the problem with padding factor $O(1)$ requires $\Omega(\log^* n)$ expected time on an n-processor standard PRAM. We show that this bound is tight:

Theorem: *n random numbers drawn independently from the uniform distribution over $(0, 1]$ can be padded-sorted in $\tilde{O}(\log^* n)$ time using $O(n)$ operations.*

We again show a tradeoff between the run time t and the padding factor λ. In one extreme, for $t = \Theta(\log^* n)$, the padding factor is larger than that of [26, 27], but still $o(1)$. Another interesting case, $t = \Theta(\log\log n/\log\log\log n)$, yields a padding factor better than that of [27], and equaling that of [20]. Our algorithm for sorting random numbers uses a generalization of the comparison-based padded-sorting algorithm to the *ordered interval allocation* problem, which may be of interest in its own right.

1.3 Integer Sorting

The problem of sorting n integers in the range $1..n$ is solved in $O(n)$ time by sequential bucket sorting. Rajasekaran and Reif [33] gave a parallel algorithm that uses $\tilde{O}(\log n)$ time and $O(n)$ work. Later improvements yielded $\tilde{O}(\log n/\log\log n)$ time with $O(n)$ work [35, 19, 30]. Hagerup [20] showed that small integers can be chain-sorted in $\tilde{O}((\log^* n)^2)$ time with $O(n)$ work. This was subsequently improved to $\tilde{O}(\log^* n)$ time and $O(n)$ work [15, 21]. Note that it is not obvious how to convert a chain-sorted output to even a padded-sorted one faster than in $\Theta(\log n/\log\log n)$ time. We show:

Theorem: *n integers in the range $1..n$ can be padded-sorted in $\tilde{O}(1)$ time on an n-processor OR PRAM, and in $\tilde{O}(\log\log n)$ time with $O(n)$ work on a standard PRAM.*

A sorting algorithm is *stable* if keys with equal values appear in the same order in the output as in the input. An algorithm that sorts n keys in the range $1..n$ stably can, using the principle of radix sorting, be turned into an algorithm that sorts n keys in the range $1..n^c$, for arbitrary fixed $c \in \mathbb{N}$, which makes stability a very desirable property. Bucket sorting is stable and so the stable and unstable versions of integer sorting have the same sequential complexity. A different situation seems to prevail in the parallel setting: The best known time-processor product for stable integer sorting in polylogarithmic time is $\Theta(n \log\log n)$; it was achieved with $O(\log n)$ run time in [17], and with $O(\log n/\log\log n)$ run time in [9] (see also [29, 34]). We give a very fast padded-sorting algorithm for this problem:

Theorem: *n integers in the range $1..n$ can be stably padded-sorted in $\tilde{O}(\log\log n/\log k)$ time on an OR PRAM with kn processors, for any k with $2 \leq k \leq \log n$, and in $O(\log\log n)$ time on an n-processor standard PRAM.*

Thus, in particular, we achieve a constant run time with $\Theta(n(\log n)^\epsilon)$ processors, for arbitrary fixed $\epsilon > 0$. A run time of $O(\log\log n/\log k)$ is natural in view of the way the problem has been approached [17, 9, 29, 34]: the bottleneck in each case is a binary search on a path of length $\Theta(\log n)$ that must be performed for every key, thus leading to the $\Theta(n\log\log n)$ work bound. The bottleneck in our case is a k-ary search on a similar path, which requires more work than a binary search.

1.4 Approximate Prefix Summation

The time complexity of exact prefix summation of n integers of $O(\log n)$ bits each on a CRCW PRAM with $n^{O(1)}$ processors is $\Theta(\log n/\log\log n)$. We consider the following *approximate prefix summation problem*: Given n nonnegative integers $x_1, \ldots, x_n$, compute n approximate prefix sums $y_1, \ldots, y_n$ such that with $y_0 = 0$, the following holds for $i = 1, \ldots, n$ and for some $\lambda \geq 0$ with $\lambda = o(1)$ called the *accuracy* of the approximation:

(1) $y_i \geq y_{i-1} + x_i$;
(2) $y_i \leq (1+\lambda)\sum_{j=1}^{i} x_j$.

Note that conditions (1) and (2) together are stronger than simply requiring each approximate prefix sum to be close to its "true value", which would allow distinct y_i's to be computed independently and could not even guarantee the monotonicity of the sequence $y_1, \ldots, y_n$. We show:

Theorem: *Approximate prefix summation problems of size n can be solved in $\tilde{O}(1)$ time on an n-processor OR PRAM, and in $\tilde{O}(\log\log n)$ time with $O(n)$ work on a standard PRAM. The constant-time result requires the input numbers to be of $O(\log n)$ bits each.*

No results whatsoever on approximate prefix summation were previously known.

2 Preliminaries

A basic fact [33, 13, 22] is that the prefix sums of n integers of $O(\log n)$ bits each can be computed in $O(\lceil n/p + \log n/\log\log p\rceil)$ time with p processors, for all integers $n \geq 2$ and $p \geq 4$. We also make frequent use of the procedure characterized in the following lemma, which we term "Ragde compaction".

Lemma 2.1: Given an array of size n containing b objects in distinct cells (b is unknown), n processors using $O(n)$ space can in constant time move the objects to distinct cells in an array of size at most b^5.

Proof: Ragde [32] showed that the objects can be compacted to an array of size at most k^4, where k is a known upper bound on b. We use this as follows: First attempt to compact the elements to an array of size k^4, where $k = \lfloor n^{1/5} \rfloor$. If this succeeds, we have enough processors to try simultaneous compaction into $\Theta(n^{1/5})$ arrays of different sizes. It is easy to choose the $\Theta(n^{1/5})$ sizes such that returning the smallest array into which the objects fit satisfies the condition of the lemma. If the compaction into k^4 space fails, on the other hand, $b \geq n^{1/5}$, and the original array can be returned. ∎

Other crucial subroutines are drawn from the "log-star tool kit" that was developed in a series of papers [30, 20, 5, 15, 16, 21, 6]. One of the most frequently used algorithms is for *semisorting*, i.e., given n integers in the range $1..n$, place these in (distinct cells of) an array of size $O(n)$ such that each value in the range $1..n$ occurs only in a subarray that does not overlap the subarray of any other value (i.e., duplicates are brought together). By Lemma 2.1, we may and will assume that the subarray of a value of multiplicity b is of size at most b^5. Semisorting can be done in $\tilde{O}(\log^* n)$ time with $O(n)$ work on a standard PRAM [21]. It can also be done in $\tilde{O}(1)$ time either on a standard PRAM with $O(n \log^{(l)} n)$ processors, for arbitrary fixed $l \in \mathbb{N}$, or on an n-processor OR PRAM. Other algorithms in the log-star family exhibit the same run times on the various models. Two such algorithms are for the problem of *interval allocation*, i.e., place n intervals of total length s (s is unknown) in nonoverlapping positions in a base interval of length $O(s)$, and for the closely related *processor allocation* problem, i.e., supply each of n requesting processors with any desired number of auxiliary processors from a pool of available processors. The special case of interval allocation in which all intervals to be placed are of length 0 or 1 is called *linear approximate compaction.*

In the following, let U be a fixed ordered universe (i.e., a set equipped with a total order "$\leq$"). Given any finite nonempty set $B \subseteq U$, let $Pred(x, B) = \max\{y \in B \mid y \leq x\}$ and $Pred^+(x, B) = \max\{y \in B \mid y < x\}$, for all $x \in U$ for which these quantities are defined. B partitions any finite subset A of U into subsets called the *segments* induced by B in A or the segments of B in A; two elements $x, y \in A$ belong to the same segment if and only if $Pred(x, B) = Pred(y, B)$, except that we consider elements in A smaller than $\min B$ to belong to the same segment as those elements in A larger than or equal to $\max B$ (i.e., we allow "wrap-around"). In particular, if $|B| = 1$ there is just one segment consisting of all elements in A. To *locate* A with respect to B is to label each element of A with the segment induced by B to which it belongs (represented in the obvious way by an element of B), and to *partition* A with respect to B is to semisort the elements of A with respect to these labels. The *maximum* and *minimum gap* of B in A, $maxgap(B, A)$ and $mingap(B, A)$, are defined as the maximum and minimum cardinality, respectively, of a segment of B in A. If $A \neq \emptyset$, define the *spread* of B in A as the real number $\lambda \geq 0$ with

$$\frac{maxgap(B, A)}{mingap(B, A)} = 1 + \lambda.$$

Note that λ is a measure of how well B splits A into segments of approximately equal sizes. Much of the present paper is concerned with computing samples of low spread of a given ground set. We now discuss two special kinds of samples. A *naive sample* B of a set A of size $m \leq |A|$ is obtained by letting each of m processors choose an element at random from the uniform distribution over A, with distinct processors acting independently. Note that the sample B may be a multiset; if we use it in a context where only totally ordered sets are meaningful, an arbitrary ordering among elements of the same value is to be assumed. If $A \neq \emptyset$ and $g \in \mathbb{N}$, define a *g-regular sample* of A as a subset B of A with $g \leq mingap(B, A) \leq maxgap(B, A) \leq g + 1$ that includes the first element of A.

Lemma 2.2: For every $g \in \mathbb{N}$, every finite ordered set A of size at least g^2 has a g-regular sample. Furthermore, if each element of A is marked with its rank in A and has an associated processor, a g-regular sample of A can be computed in constant time using constant space per processor. ∎

By a *predecessor structure* for a nonempty set $S \subseteq U$ we mean a data structure that supports *Pred* and $Pred^+$ queries on S, i.e., is able to return $Pred(x, S)$ and $Pred^+(x, S)$ for arbitrary given $x \in U$ (with a special value "undefined" returned as appropriate).

Lemma 2.3:

(a) For all given integers n and k with $2 \leq k \leq n$, a predecessor structure for n keys drawn from an ordered universe that can be queried in $O(\log n/\log k)$ time with k processors can be constructed using $O(\log n/\log k)$ time, kn processors and $O(kn)$ space with probability at least $1/2$;

(b) There is a constant $\epsilon > 0$ such that for every fixed $c \in \mathbb{N}$ and for all integers $n \geq 4$, a predecessor structure for n keys drawn from $1..n^c$ that can be queried in $O(\log\log n/\log k)$ time with k processors, for any integer k with $2 \leq k \leq \log n$, can be constructed using constant time, $O(n \log n)$ processors and $O(n \log n)$ space with probability at least $1 - 2^{-n^\epsilon}$;

(c) For all integers $n \geq 2$, a predecessor structure for n keys drawn from $1..n$ that can be queried in constant time with one processor can be constructed in constant time either on a standard PRAM with $O(n \log n)$ processors and $O(n \log n)$ space, or on an n-processor OR PRAM with $O(n)$ space.

Proof: (a) Assume that $k \geq (\log n)^2$, since otherwise we can simply sort the keys using the algorithm of Rajasekaran and Reif [33] and execute a query by means of k-ary search. We can also assume that we have k^2 processors per key and that a query is executed with k^2 processors, since the time bounds of $O(\log n/\log k)$ are insensitive to the substitution of $\sqrt{k}$ for k.

Our approach is to construct a random search tree of degree $k+1$ over the input set. Initially the tree consists only of its root, and all keys are located at the root. In general, if m keys are present at a node u, we allow these to be stored in an array Q of size $O(m)$, and the tree construction proceeds from u as follows: Each of k processors chooses a random location in Q. A multiset S is then formed from the keys found in these locations, with dummy keys representing empty locations, and $k+1$ children of u are created, each of which corresponds to one of the segments of the universe induced by S. Some children will correspond to dummy keys and be useless. Except with negligible probability, however, the number of real keys in S will be $\Omega(k)$, which suffices for our purposes. Now each key located at u uses k^2 associated processors to find its predecessor in S, after which it is moved to the appropriate child of u. In order to move the keys to the appropriate children, we first semisort them by their destination nodes, which can be done in constant time with $k \geq \log n$ processors per key. This stores the keys destined for a child v of u in a separate subarray Q', but we do not necessarily have $|Q'| = O(m')$, where $|Q'|$ denotes the size of Q' and m' is the number of keys to be placed at v. We therefore use an algorithm for linear approximate compaction to simultaneously attempt to compact the keys in Q' into arrays of sizes $1, 2, 4, \ldots, 2^{\lfloor \log |Q'| \rfloor}$, with $\Theta(\log n)$ trials for each size, and then select the smallest array for which the compaction succeeded, or Q' itself if no compaction succeeded. The size of the array so chosen is $O(m')$, except with negligible probability.

Once the keys of a node u are stored in an array of size at most k, S can instead be chosen as the set of all keys, and the children of u become leaves in the search tree. It can be shown that except with negligible probability, the height of the search tree will be $O(\log n/\log k)$. Finally observe that a query can search in the tree essentially as described above.

Parts (b) and (c) were shown by Raman [34] and by Berkman and Vishkin [8], respectively. Meeting the space bound of part (b) involves using the perfect hashing of [6]. ∎

When speaking about (padded-)sorting a set of keys, we assume that each key is just one field of a record containing arbitrary associated information, and we require of a (padded-)sorting algorithm that it is able to move the entire input records or pointers to them to the output array (rather than simply recreating the keys there).

3 Comparison-Based Sorting

We show in this section that the comparison-tree time bound of $\Theta(\log n/\log k)$ for sorting with kn processors, where $2 \leq k \leq n$, can be matched exactly on a randomized PRAM. In fact, we solve the related problem of *approximate ranking*, defined as follows: Given n keys drawn from an ordered universe and $\lambda \geq 0$, label each key with a rational number such that if r_i is the label of the key of rank i, for $i = 1, \ldots, n$, and $r_0 = 0$, then $\frac{1}{n} \leq r_i - r_{i-1} \leq \frac{1+\lambda}{n}$, for $i = 1, \ldots, n$ (if several keys are identical, the ranks $1, \ldots, n$ must be assigned to the n keys in a way consistent with any stability requirement). λ is called the *ranking accuracy.*

As demonstrated in the following simple lemma, going from approximate ranking with accuracy λ to padded sorting with padding factor λ is a matter of multiplying by n and rounding.

Lemma 3.1: Let $n \in \mathbb{N}$ and let $r_0, \ldots, r_n$ be real numbers with $r_0 = 0$ and $\frac{1}{n} \leq r_i - r_{i-1} \leq \frac{1+\lambda}{n}$, for $i = 1, \ldots, n$. For $i = 0, \ldots, n$, let $y_i = \lfloor nr_i \rfloor$. Then

(a) For $i = 1, \ldots, n$, $y_i > y_{i-1}$;

(b) For $i = 1, \ldots, n$, $y_i \leq (1+\lambda)i$;

(c) For $0 \leq i \leq j \leq n$, $y_j - y_i \leq \lceil (1+\lambda)(j-i) \rceil$. ∎

Our algorithm is best described as storing the input in an (a, b)-tree T, where a and b are large and b/a is close to 1. The n input keys are hence stored in order from left to right in the n leaves of T, all of which are at the same depth h, and the degree of each internal node in T lies in the set $\{a, \ldots, b\}$ (this is not quite exact; see below). Associate a subinterval of the unit interval $[0, 1]$ with each node of T as follows. The root of T is associated with the full interval $[0, 1]$, and the interval $[\alpha, \beta]$ associated with a nonleaf node u of degree d is partitioned into d equal-sized subintervals, the ith of which, $[\alpha + \frac{i-1}{d}(\beta - \alpha), \alpha + \frac{i}{d}(\beta - \alpha)]$, is associated with the ith child of u, for $i = 1, \ldots, d$. Since the interval associated with a leaf results from exactly h splittings into between a and b subintervals, it is clear that the length of each such leaf interval lies between b^{-h} and a^{-h}. Furthermore, $a^h \leq n \leq b^h$. Hence for any λ with $1 + \lambda/3 \geq (b/a)^h$, the length of each leaf interval lies between $\frac{1}{n(1+\lambda/3)}$ and $\frac{1+\lambda/3}{n}$. Thus if we define r_i as $1 + \lambda/3$ times the right endpoint of the interval associated with the ith leaf of T, for $i = 1, \ldots, n$, and take $r_0 = 0$, then for $i = 1, \ldots, n$, $\frac{1}{n} \leq r_i - r_{i-1} \leq \frac{(1+\lambda/3)^2}{n} \leq \frac{1+\lambda}{n}$, provided that $\lambda \leq 1$, i.e., we have solved the approximate ranking problem with accuracy λ. Note that the above argument remains valid even if the node degrees in T do not lie between a and b, as long as the ratio between the degrees of any two internal nodes in T on the same level is bounded by b/a. We use this to allow nodes at height 1 to have more than b children.

T is constructed as follows: For an appropriate integer $g \geq 5$ we compute samples $S_0, \ldots, S_h$ of the input set X. $S_h = X$, and for $i = 0, \ldots, h-1$, $g^i \frac{1}{1+1/g} \leq |S_i| \leq g^i$, and S_i has spread at most $1/g$ in X and includes the smallest element x_1 of X. For $i = 0, \ldots, h$, we identify the nodes in T at depth i with S_i and define the parent of a node x at depth $i \geq 1$ as $Pred(x, S_{i-1})$ (because $x_1 \in S_{i-1}$, this is well-defined). It is now easy to see that the degree of every node in

T at depth $i \leq h-2$ is at most

$$\left\lceil \frac{maxgap(S_i, X)}{mingap(S_{i+1}, X)} \right\rceil \leq \left\lceil \frac{n(1+1/g)^2/g^i}{n/(g^{i+1}(1+1/g))} \right\rceil \leq g+4$$

and, similarly, at least $g-4$. Furthermore, since the spread of S_{h-1} in S_h is at most $1/g$, the ratios of the degrees of two nodes in T at height 1 is at most $1+1/g$. We have hence realized the scheme outlined above with $a = g-4$ and $b = g+4$.

We do not know how to compute $S_0, \ldots, S_h$ deterministically. Furthermore, naive random sampling falls far short of providing the required low spread. We instead use a more sophisticated sampling due to Reif and Valiant [36, Lemma 7.1], the basic idea of which is very simple: to obtain a sample of size s, pick a naive sample of size $s \log n$ rather than s, then keep only every $(\log n)$th element of the sample. The resulting s keys are far more evenly spaced than if they had been chosen directly using naive sampling. Whereas Reif and Valiant keep sample elements that are exactly equally-spaced in the original larger sample, this is impossible in our setting, and we have to generalize their approach somewhat.

Lemma 3.2: Let A be a nonempty finite ordered set and let g be a positive integer with $g \geq \log |A|$. Let B be a naive sample of A and let C be an arbitrary subset of B such that the spread of C in B is at most $1/g$ and the minimum gap of C in B is at least $192g^3$. Then the spread of C in A exceeds $8/g$ with probability at most 2^{-g}.

Proof: If the spread of C in A exceeds $8/g$, then either $maxgap(C, A) \geq \frac{|A|}{|C|}\sqrt{1+8/g} \geq \frac{|A|}{|C|}(1+2/g)$ or $mingap(C, A) \leq \frac{|A|}{|C|}(1+2/g)^{-1}$. If $maxgap(C, A) \geq \frac{|A|}{|C|}(1+2/g)$, then some contiguous segment D of A (allowing "wrap-around") of size at least $\frac{|A|}{|C|}(1+2/g)-1$ contains no elements of C, and therefore fewer than $maxgap(C, B)$ elements of B. The number N of elements of B falling into D is binomially distributed with expected value

$$\begin{aligned}\frac{|B||D|}{|A|} &\geq \frac{|B|}{|A|}\left(\frac{|A|}{|C|}(1+2/g)-1\right) \geq \frac{|B|}{|C|}(1+2/g)-1 \\ &\geq \frac{maxgap(C,B)}{1+1/g}(1+2/g)-1 \geq \frac{maxgap(C,B)}{1-1/(4g)}.\end{aligned}$$

Now by the Chernoff bound $\Pr(N \leq (1-\epsilon)E(N)) \leq e^{-\epsilon^2 E(N)/2}$, used with $\epsilon = 1/(4g)$, the above event happens for each fixed set D with probability at most $e^{-192g^3/(32g^2)} \leq e^{-4g}$. Summing over the at most $|A|^2$ choices of D yields $\Pr(maxgap(C, A) \geq \frac{|A|}{|C|}(1+2/g)) \leq |A|^2 \cdot e^{-4g} \leq \frac{1}{2}|A|^2 \cdot 2^{-2g} \cdot 2^{-g} \leq \frac{1}{2} \cdot 2^{-g}$. Likewise, $\Pr(mingap(C, A) \leq \frac{|A|}{|C|}(1+2/g)^{-1}) \leq \frac{1}{2} \cdot 2^{-g}$. ∎

Our actual sampling procedure is described in the following lemma.

Lemma 3.3: Let n, k, t, s and g be given positive integers with $n, k \geq 4$, $t \geq \lceil \log n / \log k \rceil$, $\log n \leq g \leq 2^{t \log\log(kn)}$ and $sg^{64} \leq n$. Then, given an ordered set A of size n, a sample C of A with $s \cdot \frac{1}{1+1/g} \leq |C| \leq s$ that includes the first element of A and has spread at most $1/g$ in A can be computed using $O(t)$ time, kn processors and $O(kn)$ space with probability at least $1-2^{-g}$.

Proof: We show only a weaker statement that does not require C to contain the first element of A. Observe first that a sample C of the required form always exists, so that we can ignore values of n (and hence g) below any fixed constant. Let $g' = 192(8g)^3$ and assume (by the observation just made) that $s(g')^{21} \leq n$. Let B be a naive sample of A of size sg'. We now describe the main steps of how to obtain C from B, and only afterwards worry about the actual implementation of these steps.

If $s < (g')^2$, let C be an s-element subset of B of zero spread. If $s \geq (g')^2$, instead let S be a naive sample of B of size $\lfloor s/(g')^2 \rfloor$, merge each segment of B induced by S of size less than $(g')^2$ with the segment following it by removing one element from S and let $S' \subseteq S$ be the set of elements of S that survive this process. Obviously every segment of B induced by S' is of size at least $(g')^2$. Now use Lemma 2.2 to compute a g'-regular sample of each such segment and let C be the union of these g'-regular samples.

In either case we have obtained a sample C with $s \cdot \frac{1}{1+1/g} \leq |C| \leq s$ and with spread at most $1/g'$ in B; to see this in the case $s \geq (g')^2$, it suffices to note that $g' \leq mingap(C,B) \leq maxgap(C,B) \leq g'+1$ and to recall that $|B| = sg'$. We may therefore conclude from Lemma 3.2 that the spread of C in A exceeds $8/(8g) = 1/g$ with probability at most 2^{-8g}.

We now show how to carry out the procedure described above for the case $s \geq (g')^2$; it will become obvious how to handle the easier case $s < (g')^2$. It is easy to see that it suffices to describe how to execute the following steps, first with $R = S$ and afterwards with $R = S'$:

(1) Partition B according to R;

(2) Sort the elements of each segment of B induced by R (without padding); in particular, this computes the size of each segment.

We implement Step (1) using Lemma 2.3(a). By executing $2g'$ independent attempts to construct the predecessor structure of Lemma 2.3(a) and to carry out the subsequent semisorting, we can ensure that the failure probability is at most $2^{-g'}$.

In Step (2) we will assume that each segment to be sorted is of size at most $2(g')^4$. If this is not the case, some $(g')^4$ successive elements of B comprise either at least $(g')^2$ elements of S (none of which are included in S'), or none at all. The expected number of elements of S among any $(g')^4$ fixed elements of B being close to g', an application of Chernoff's bounds easily shows the assumption to be violated with probability at most 2^{-8g} (say). Since we have at least $k(g')^{20}$ processors for each element of B, it is easy to sort a segment of B of size at most $2(g')^4$, and therefore stored in an array of size $O((g')^{20})$, by comparing every pair of elements in the segment and obtaining the rank of each element as one more than the number of smaller elements. The number of smaller elements is counted using prefix summation, which, given the $\Omega(k)$ processors per element, runs in time

$$O\left(\frac{\log g}{\log\log(kg)}\right) = O\left(\frac{t\log\log(kn)}{\log\log k + \log\log g}\right) = O\left(\frac{t\log\log(kn)}{2\log\log k + \log t}\right)$$
$$= O\left(\frac{t\log\log(kn)}{\log\log k + \log(t\log k)}\right) = O\left(\frac{t\log\log(kn)}{\log\log k + \log\log n}\right) = O(t).$$

Finally note that the various failure probabilities incurred add up to less than 2^{-g}. ∎

Theorem 3.4: There is a constant $\epsilon > 0$ such that for all given integers $n, k \geq 2$ and $t \geq \lceil \log n/\log k \rceil$, n elements drawn from an ordered universe can be approximately ranked with accuracy $2^{-t\log\log(kn)}$ using $O(t)$ time, kn processors and $O(kn)$ space with probability at least $1 - 2^{-n^\epsilon}$.

Proof: Obtaining the very small failure probability of 2^{-n^ϵ} requires some additional ideas not explained here. We describe a simpler algorithm with a larger (but still small) failure probability and in fact, in the interest of brevity, omit all references to actual probabilities. The algorithm furthermore achieves an accuracy of $2^{-t\log\log k}$ only, the full accuracy of $2^{-t\log\log(kn)}$ being reached via a recursive application. As in the proof of Lemma 2.3(a), we can assume that $k \geq \log n$ and that each input key has k^2 associated processors.

Let $g = 48 \cdot 2^{t\lceil \log\log k \rceil}$. If $g^{64} > n$, the input can be sorted in $O(t)$ time using the algorithm of Cole [12]. Otherwise let $h \geq 1$ be the largest integer with $g^{h-1}g^{64} \leq n$. The algorithm consists of the following steps:

(1) Construct the tree T, i.e.,

(1.a) Compute the sets $S_0, \ldots, S_h$;

(1.b) Determine the parent of each node in $\bigcup_{i=1}^{h} S_i$;

(1.c) Sort each set of siblings in T;

(2) Determine the ancestors of each leaf in T;

(3) Compute the interval associated with each leaf.

The steps are implemented as follows:

Step (1.a): In order to obtain $S_0, \ldots, S_{h-1}$, simply apply Lemma 3.3 with $s = 1, g, \ldots, g^{h-1}$ (in parallel), which requires $hkn \leq k^2 n$ processors and $O(t)$ time.

Step (1.b): By definition, our task is to locate S_{i+1} with respect to S_i, for $i = 0, \ldots, h-1$. We therefore use Lemma 2.3(a) to construct a predecessor structure for S_i, for $i = 0, \ldots, h-1$. Since $\sum_{i=0}^{h-1} |S_i| \leq \sum_{i=0}^{h-1} g^i \leq g^h \leq n/g^{63}$, we can attempt each construction sufficiently many times in parallel to achieve a high reliability. The construction and the subsequent search take $O(\lceil \log n / \log k \rceil) = O(t)$ time.

Step (1.c): After partitioning S_{i+1} with respect to S_i using the predecessor structures constructed in Step (1.b), for $i = 0, \ldots, h-1$, we sort each set of siblings using the algorithm of Cole [12]. Since the degree of each internal node in T is $g^{O(1)}$ (note, in particular, that this holds for the nodes at level 1), the time needed is $O(\log g / \log\log k) = O(t)$.

Step (2): For each leaf v in T and for $i = 0, \ldots, h$, let $p_i(v)$ be the ancestor of v in T at depth i and note that there is a "natural candidate" for $p_i(v)$, namely $u = \mathit{Pred}(v, S_i)$. We may in fact have $p_i(v) \neq u$, but then $p_i(v) = \mathit{Pred}^+(u, S_i)$. The proof of this fact, informally, bounds how far the ancestors of v can get from v. In precise terms, let ρ_j be the rank of $p_j(v)$ in the input set X, for $j = 0, \ldots, h$. Then clearly $0 \leq \rho_{j+1} - \rho_j \leq \mathit{maxgap}(S_j, X)$, for $j = 0, \ldots, h-1$, and hence if $i < h$, $0 \leq \rho_h - \rho_{i+1} \leq \sum_{j=i+1}^{h-1} \mathit{maxgap}(S_j, X) \leq \mathit{mingap}(S_i, X)$, from which the claim follows.

Using the predecessor structures constructed in Step (1.b), each leaf v in T can easily determine its two candidate ancestors at each level. This leaves only 2^h candidates for the ancestor sequence $(p_h(v) = v, p_{h-1}(v), \ldots, p_0(v))$. Each candidate sequence $(u_h = v, u_{h-1}, \ldots, u_0)$ can be checked in constant time by h processors, who simply verify that u_{j-1} is indeed the parent of u_j, for $j = 1, \ldots, h$. Using $h \cdot 2^h$ processors, v can therefore check all candidate sequences in parallel and determine the correct sequence of its ancestors in constant time. Since $h = O(\log n / \log g) = O(\log k / \log\log k)$, this uses $o(k)$ processors per leaf.

Step (3): Label each node in T in the following way with a subinterval of $[0,1]$ called its *immediate interval*: The immediate interval of the root of T is $[0,1]$, and the immediate interval of the ith child of a node of degree d is $[\frac{i-1}{d}, \frac{i}{d}]$. The immediate intervals of the nodes in T should not be confused with the intervals introduced previously, which we call their *compounded intervals*. Given two subintervals I_1 and I_2 of $[0,1]$, define their *composition*, $I_1 \circ I_2$, as the interval obtained by scaling and translating I_2 in a way that would transform $[0,1]$ to I_1, i.e., $[\alpha_1, \beta_1] \circ [\alpha_2, \beta_2] = [\alpha_1 + \alpha_2(\beta_1 - \alpha_1), \alpha_1 + \beta_2(\beta_1 - \alpha_1)]$. The operation $\circ$ is associative, and it is easy to see that the compounded interval of a leaf v is $[\alpha_0, \beta_0] \circ \cdots \circ [\alpha_h, \beta_h]$, where $[\alpha_i, \beta_i]$ is the immediate interval of the ancestor of v at depth i, for $i = 0, \ldots, h$. Our task is hence to

compose the immediate intervals on every root-to-leaf path in T, which we do independently for each such path.

Let us first look at the size of the objects involved. We represent an interval in the obvious way by a pair of two rational numbers, each of which is in turn represented by two integers. For every $j \in \mathbb{N}$, the endpoints of the composition of j immediate intervals can be expressed with denominators of size $g^{O(j)}$, and the naive algorithm for computing the composition (i.e., not attempting to reduce fractions) creates no larger integers. We can hence fix a constant $c \in \mathbb{N}$ such that the composition of j immediate intervals can be represented in $cj \log g$ bits, for every $j \in \mathbb{N}$. In particular, every compounded interval can be represented in $O(h \log g) = O(\log n)$ bits, which implies that all relevant operations on the intervals manipulated by the algorithm can be executed in constant time.

For $k < g^{2c}$, it is therefore easy to compute the compounded interval of a leaf v: One processor simply follows the path in T from the root to v, composing immediate intervals as it goes along, which takes $O(h) = O(t)$ time. For $k \geq g^{2c}$, let $j \in \mathbb{N}$ be maximal with $cj \log g \leq \frac{1}{2} \log k$ and note that $j = \Omega(\log k / \log g)$. We compose groups of $\min\{j, h+1\}$ consecutive immediate intervals on the path in T from the root to v in constant time using table lookup. This leaves $O(\lceil h/j \rceil) = O(t)$ intervals that can be processed sequentially as before. In more detail, since the representations of j immediate intervals involve altogether at most $\frac{1}{2} \log k$ bits, these can easily be stored together in one word by $O(j\sqrt{k}) = o(k)$ processors. This word is then converted to the composition of the j intervals by lookup in a table, which, being of size $O(\sqrt{k})$, can be constructed in constant time by k processors; we omit the details. The total number of processors needed per leaf in T is $o(hk) = o(k^2)$.

As follows from calculations carried out in the beginning of this section, the algorithm achieves an accuracy of λ, where $1+\lambda/3 = \frac{g+4}{g-4} = 1+\frac{8}{g-4} \leq 1+\frac{16}{g}$, i.e., $\lambda \leq 48/g \leq 2^{-t \log\log k}$. ∎

Corollary 3.5: There is a constant $\epsilon > 0$ such that for all given integers $n, k \geq 2$ and $t \geq \lceil \log n / \log k \rceil$, n elements drawn from an ordered universe can be padded-sorted with padding factor $2^{-t \log\log(kn)}$ using $O(t)$ time, kn processors and $O(kn)$ space with probability at least $1 - 2^{-n^\epsilon}$. ∎

The corollary follows immediately from Lemma 3.1. Note the following remarkable consequence of part (c) of that lemma: Any run of $\lceil 1/\lambda \rceil$ consecutive input keys are stored in consecutive locations of the output array, except that there may be a single empty cell between the first and the last element of the run.

We finally consider a generalization of padded sorting called *ordered interval allocation*. The problem here is, given n keys $x_1, \ldots, x_n$ with associated nonnegative integer demands $d_1, \ldots, d_n$ and a padding factor $\lambda \geq 0$, to compute nonoverlapping intervals $I_1, \ldots, I_n$ of sizes $d_1, \ldots, d_n$ within a base interval of size at most $(1+\lambda) \sum_{j=1}^{n} d_j$ such that for all $i, j \in \{1, \ldots, n\}$, if $x_i < x_j$, then I_i is to the left of I_j. The following theorem shows that ordered interval allocation is no harder than padded sorting.

Theorem 3.6: There is a constant $\epsilon > 0$ such that for all given integers $n, k \geq 2$ and $t \geq \lceil \log n / \log k \rceil$, ordered interval allocation problems of size n with padding factor $2^{-t \log\log(kn)}$ can be solved using $O(t)$ time, kn processors and $O(kn)$ space with probability at least $1 - 2^{-n^\epsilon}$.

Proof: Let the input keys and their demands be $x_1, \ldots, x_n$ and $d_1, \ldots, d_n$, respectively, and take $\lambda = 2^{-t\lceil \log\log(kn) \rceil}$ and $D = \sum_{j=1}^{n} d_j$. A simple solution to the problem would be to padded-sort a multiset of D keys where x_j occurs d_j times, for $j = 1, \ldots, n$, using the algorithm of Corollary 3.5. This procedure, referred to below as the *naive algorithm*, cannot be used directly, since the number of operations needed would be at least proportional to D, on which we have

placed no bound. Since a relative accuracy of λ is all that is needed, however, we can scale down the demands, solve the problem for the smaller demands, and scale back up the solution.

For every $r \geq 1$, if we replace d_j by $\lceil (d_j + 1)/r \rceil$, for $j = 1, \ldots, n$, solve the resulting ordered interval allocation problem with padding factor μ and scale up the solution by replacing each interval endpoint y by $\lfloor ry \rfloor$, then the scaled-up intervals are nonoverlapping and of at least the required sizes. The sum of the scaled-down demands is at most $\frac{D+n}{r} + n$, and the size of the scaled-up base interval will therefore be at most $r(1+\mu)(\frac{D+n}{r} + n) = (1+\mu)(D + (r+1)n)$.

We first use the algorithm of Corollary 3.5 to padded-sort the keys with padding factor 1. Let $g = 1/\lambda$ and divide the padded-sorted output array into $\lceil 2\lambda n \rceil$ *segments* of at most g cells each. We then scale the demands within each segment so that the scaled values are in the range $0..8g^2 + 1$ and perform ordered interval allocation inside the segment using ordinary prefix summation. In more detail, consider any fixed segment, let m be the maximum demand in this segment and let s be the sum of the demands in the segment. If $m \geq 8g^2$, scale the demands in the segment as described above with $r = m/(8g^2)$; otherwise use the original demands. In either case the allocation within the segment is now done exactly (i.e., $\mu = 0$) by means of prefix summation; all demands can be satisfied with a base interval of size s if no scaling was applied, and otherwise of size at most $s + (r+1)g \leq s + 2rg = s + m/(4g) \leq (1 + \lambda/4)s$. The prefix summation is of g values of $O(\log g)$ bits each and can therefore be carried out in time $O(\log g / \log\log(kg)) = O(t)$. Since m can be computed in constant time if $k \geq g$ and in $O(\log\log g)$ time otherwise, this takes no more time.

We have now reduced the problem to one with $\lceil 2\lambda n \rceil$ keys. If $\lceil 2\lambda n \rceil = 1$, we are done. Hence assume that the number of keys in the reduced problem is bounded by $4\lambda n$ and let $D' \leq (1+\lambda/4)D$ be the sum of the demands in the reduced problem. We again scale the demands by an appropriate number r and solve the resulting problem with padding factor $\lambda/4$. It suffices to choose r so that the size of the scaled-up base interval is at most $(1 + \lambda/2)D' \leq (1+\lambda)D$, which is guaranteed if $(1 + \lambda/4)(D' + (r+1)4\lambda n) \leq (1 + \lambda/2)D'$, and hence if $32(r+1)n \leq D'$. Compute an approximation $\widehat{D}$ to D' with $D' \leq \widehat{D} \leq 2D'$ (see [20, 21]). If $\widehat{D} \geq 128n$, we can take $r = \frac{\widehat{D}}{64n} - 1 \leq \frac{D'}{32n} - 1$, which is small enough, while the sum of the scaled-down demands is at most $\frac{D'+n}{\widehat{D}/(128n)} + n \leq 130n$. These considerations reduce the original problem to one with total demand $O(n)$, which can be solved using the naive algorithm above. The only nontrivial implementation problem, how to create the multiset of keys to be sorted, can be solved using ordinary interval allocation. ∎

4 The Lower Bound

Sarnath [38] recently showed that the lower bound of Håstad [23] can be modified to yield a lower bound for padded sorting. Translated to the PRAM setting, his result is that any (deterministic) algorithm that runs on a CRCW PRAM with $n^{O(1)}$ processors and padded-sorts n bits into an output array of size $n + n^{\epsilon} - 1$, where $0 < \epsilon < 1$, has a run time of $\Omega\left(\frac{(1-\epsilon)\log n}{\log\log n + c}\right)$, for some constant c. In this section we give a simpler proof of the lower bound and render it in a form that brings out more clearly its relationship to our upper bounds. A still simpler proof was given by Håstad [24].

The lower bound extends to randomized algorithms, as we show using standard techniques. Although the proof below assumes the given padded-sorting algorithm to be a *Las Vegas* algorithm, which is always correct, the result also holds for *Monte Carlo* algorithms, which work in a fixed time, but may err (with probability bounded by a constant smaller than 1). This is

because any Monte Carlo algorithm for padded sorting with run time t can be turned into a Las Vegas algorithm with expected run time $O(t)$ — simply execute the Monte Carlo algorithm repeatedly until its output is correct.

Theorem 4.1: Suppose that a Las Vegas algorithm padded-sorts $n \geq 2$ bits with padding factor $\lambda \geq 1/n$ in expected time t on a p-processor OR PRAM (with a word length of $O(\log(n+p))$ bits), where $p \geq 4$. Then $\lambda = 2^{-O(t \log\log p)}$.

Proof: If $p < \sqrt{n}$, the claim is trivial, since $t = \Omega(\sqrt{n})$. On the other hand, for $p \geq \sqrt{n}$ we have $\log\log p = \Omega(\log\log n)$. Assume hence that $p \geq n^2$. We also assume that $\lambda \leq 1/(2d)$, where d is a constant to be fixed below.

For all integers $n \geq 1$ and $m \geq 0$, consider the following (n,m)-*counting* problem: Given n bits $x_1, \ldots, x_n$, compute an integer r with $r \leq \sum_{j=1}^{n} x_j \leq r + m$. Informally, the (n,m)-counting problem is to count the number of 1's among the input bits with an error of at most m. We will show by induction on i that there are constants $c, d \in \mathbb{N}$ such that $P(i)$ holds for all integers $i \geq 0$, where $P(i)$ is the assertion

> $(n, \lfloor n/g^i \rfloor)$-counting problems can be solved in expected time
> at most $c(i+1)t$ on a p-processor OR PRAM,

and $g = \lfloor 1/(d\lambda) \rfloor \geq 2$.

$P(0)$ is trivial. Assume now that $P(i)$ holds for some integer $i \geq 0$ and suppose that we are given an $(n, \lfloor n/g^{i+1} \rfloor)$-counting problem. Begin by padded-sorting the input using the algorithm assumed in the theorem, which takes expected time t. Then determine the position of the first 1 in the output, trivial to do in constant time with n^2 processors, and mark each empty cell in the output array following this first 1. Counting the number of 1's in the input with error at most m now reduces to determining the number s of marked cells in the output array with error at most m. We have $s \leq \lambda n$. Hence with the constant d chosen suitably, we can use an algorithm for linear approximate compaction to store the s marked cells in an array Q of size at most $d\lambda n$. This takes constant expected time.

Since $g \cdot d\lambda n \leq n$, we can easily derive from Q a bit array Q' of size n that contains exactly gs 1's: Simply replace each marked cell in Q by g 1's, replace each unmarked cell in Q by g 0's, and add so many 0's as to have exactly n bits. Apply the $(n, \lfloor n/g^i \rfloor)$-counting algorithm implied by $P(i)$ to Q', which produces an integer r with $r \leq gs \leq r + \lfloor n/g^i \rfloor$ and hence $r/g \leq s \leq r/g + \lfloor n/g^i \rfloor/g$. Since s is an integer,

$$\lceil r/g \rceil \leq s \leq \lfloor r/g + \lfloor n/g^i \rfloor/g \rfloor \leq \lceil r/g \rceil + \lfloor n/g^{i+1} \rfloor,$$

i.e., we have determined s with error at most $\lfloor n/g^{i+1} \rfloor$, as desired. The expected time consumed by the complete algorithm is at most $c(i+1)t + t + O(1)$, which for c chosen sufficiently large is bounded by $c(i+2)t$. This ends the inductive proof.

Let $i_0 = \lfloor \log n/\log g \rfloor + 1$ and observe that $P(i_0)$ states that $(n,0)$-counting problems can be solved with p processors in time $O(i_0 t) = O(t \log n/\log g)$. But the $(n,0)$-counting problem is simply to compute the exact sum of n bits. Hence by the lower bound of Beame and Håstad [7, Theorem 4.1(b)], the solution of $(n,0)$-counting problems requires $\Omega(\log n/\log\log p)$ time on any deterministic p-processor standard PRAM. Using the simulation mentioned in the introduction and (a simple version of) the method of Ajtai and Ben-Or [2], the bound of $\Omega(\log n/\log\log p)$ can be extended to the OR PRAM and to the expected run time of randomized algorithms, respectively. Hence $t \log n/\log g = \Omega(\log n/\log\log p)$, i.e., $\log g = O(t \log\log p)$. Since $g = \Omega(1/\lambda)$, it follows that $\lambda = 2^{-O(t \log\log p)}$. ∎

5 Sorting Random Numbers

Theorem 5.1: There is a constant $\epsilon > 0$ such that for all given integers $n \geq 2$ and $t \geq \log^* n$, n random numbers drawn independently from the uniform distribution over the interval $(0, 1]$ can be padded-sorted with padding factor t^{-t} using $O(t)$ time, $O(n)$ operations and $O(n)$ space with probability at least $1 - 2^{-n^\epsilon}$.

Proof: Let the numbers to be sorted be $z_1, \ldots, z_n$ and conceptually assign these to n buckets by taking $x_j = \lceil nz_j \rceil$, for $j = 1, \ldots, n$. For $i = 1, \ldots, n$, let B_i be the set of those input elements z_j with $x_j = i$ and take $b_i = |B_i|$. Let $\lambda = t^{-2t}$. The algorithm consists of the following steps:

(1) Semisort the input by bucket number;

(2) Independently padded-sort each bucket with padding factor λ. As a result, we have arrays $Q_1, \ldots, Q_n$ of sizes at most $(1+\lambda)b_1, \ldots, (1+\lambda)b_n$ such that Q_i contains the elements of B_i in sorted order, for $i = 1, \ldots, n$;

(3) Place (the contents of) $Q_1, \ldots, Q_n$ in that order in an array of size at most $(1 + t^{-t})n$.

Step (1) can be executed optimally in $O(t)$ time. For Step (2), we use the following:

(a) Except with negligible probability, $\max\{b_i : 1 \leq i \leq n\} \leq n^{1/20}$;

(b) Except with negligible probability, $\sum_{i=1}^n \min\{2^{b_i}, n^{1/3}\} = O(n)$.

We omit proofs of (a) and (b), but note that (a) follows easily from a Chernoff bound, and that (b) can be shown by a martingale argument. For $i = 1, \ldots, n$, let $a_i \leq b_i^5$ be the size of the subarray holding the elements of B_i after the semisorting in Step (1).

If $t > \log n$, the n buckets can be sorted exactly in $O(t)$ time using $O(\sum_{i=1}^n a_i \log(a_i + 1))$ operations; by (a) and (b), this is $O(n)$ operations, except with negligible probability.

If $t \leq \log n$, we distinguish between small and large buckets: For $i = 1, \ldots, n$, if $a_i < (\frac{1}{3}\lfloor \log n \rfloor)^5$ and hence $2^{a_i^{1/5}} \leq \min\{2^{b_i}, n^{1/3}\}$, allocate $\Theta(2^{a_i^{1/5}})$ processors to B_i and sort B_i in constant time using the algorithm of Cole [12]. If $a_i \geq (\frac{1}{3}\lfloor \log n \rfloor)^5$ and hence $n^{1/3} \leq 2^{b_i+1}$, instead allocate $\Theta(n^{1/3}/\log n)$ processors to B_i and padded-sort B_i using Corollary 3.5; by (a), this can be done in $O(t)$ time with a padding factor of $2^{-2t \log\log n} \leq \lambda$, and with $n^{\Omega(1)}$ trials for each bucket. By (b), the total number of operations executed is $O(n)$ with high probability both for small and for large buckets. This ends the description of Step (2). Step (3) is more of a challenge. Our solution can be seen as an instance of the generic log-star algorithm of [5, 15], but several new ideas are needed. We first provide an overview of the algorithm.

Recall that the goal of Step (3) is to place (the contents of) $Q_1, \ldots, Q_n$ in that order in an array of size at most $(1 + t^{-t})n$. We will disregard the actual elements stored in $Q_1, \ldots, Q_n$ and simply represent Q_i by a *brick* J_i of length equal to the size of Q_i, for $i = 1, \ldots, n$. Here a brick should be understood as an abstract one-dimensional object of a certain length that can be placed anywhere on the real line (we might have employed the word "interval", except that we already use it in a different sense). To *combine* l bricks $I_1, \ldots, I_l$ into one large brick I is to compute integer offsets $m_1 \leq \cdots \leq m_l$ such that if $I_1, \ldots, I_l$ and I are placed on the real line with the left endpoint of I_i a distance of m_i to the right of the left endpoint of I, for $i = 1, \ldots, l$, then $I_1, \ldots, I_l$ are fully contained in I and do not overlap. In this formulation Step (3) is easily seen to be a special case of ordered interval allocation. We will carry out Step (3) by gradually combining bricks into larger and larger bricks, until only a single brick remains. This brick can then be *fixed*, e.g., its left endpoint can be placed at 0. This implicitly fixes the positions of all bricks involved in the combining process, where the position of a brick is defined as the position

of its left endpoint. In order to calculate these positions explicitly, we can retrace the combining steps of the brick combination process in the opposite order. If l bricks $I_1, \ldots, I_l$ were combined into a brick I and the position of I is known, the positions of $I_1, \ldots, I_l$ can be computed simply by adding their offsets relative to I to the position of I; running the brick construction process backwards therefore in particular computes the positions of the n original bricks $J_1, \ldots, J_n$, and we are done.

Note carefully that when speaking about bricks manipulated by the algorithm to be described, we distinguish between constant-size "brick descriptors" and the true bricks, each of which is of a certain length. Objects of the two types are actually represented in the same way, but the set of applicable operations differs in the two cases. Bricks can be combined into larger bricks as described above, in which process their lengths are of prime importance. Brick descriptors may be compacted for the purpose of efficient access, but this is standard compaction of constant-size objects, and the lengths of the bricks involved are of no relevance.

We place the n original bricks $J_1, \ldots, J_n$ in order from left to right at the n leaves of a tree T of height $O(\log^* n)$. Define the *width* of a node u in T as the number of leaf descendants of u. Nodes in T at level 1 (directly above the leaves) have width v_1^{15}, where $v_1 = 1/\lambda$, and the parent of a node of width v_l^{15} has width v_{l+1}^{15}, where $v_{l+1} = 2^{v_l}$, for $l = 1, 2, \ldots$ (i.e., the degrees increase roughly exponentially as one moves away from the leaves). We here ignore certain problems related to rounding (in particular, n may not be of the form v_l^{15}, for some $l \geq 1$). Note that each node in T is associated in a natural way with a subinterval of $(0, 1]$.

Starting from the leaves of T, the algorithm attempts to combine the descendant leaf bricks of a node u at level l and of width v^{15} in left-to-right order into one large brick of length at most $(1 + \lambda)^{l+1} v^{15}$. This may or may not succeed, depending on which we call u *good* or *bad*. We actually care only about the root and want to show that it is good with high probability.

The combining at a node u is done using ordinary prefix summation for nodes at level 1 and using the ordered interval allocation algorithm of Section 3 for nodes at level ≥ 2. The latter combining is facilitated by the fact that many descendant leaf bricks have already been combined into larger bricks by the good descendants of u. This allows us to obtain enough processors per brick to make the algorithm of Theorem 3.6 run in constant time at each node.

A node u may become bad for essentially three different reasons:

(A) u may have so many bad descendants and, as a result, so many surviving descendant bricks that enough processors to make the algorithm of Theorem 3.6 run in constant time are not available ("Bad Subcontractors");

(B) Even if the number of surviving descendant bricks is small, their total length may be too large ("Act of God");

(C) Even in the best of circumstances, the randomized algorithm executed at u may fail ("Bad Luck").

The goal of the analysis is to show that a node of width v^{15} becomes bad with probability $2^{-\Omega(v^2)}$, which implies that the root is bad with negligible probability. This is easy as regards the probabilities associated with reasons (B) and (C) ("Act of God" and "Bad Luck"), since individual nodes can be analyzed independently. Because of the strong interdependence between ancestors and descendants, reason (A) ("Bad Subcontractors") is somewhat more difficult to handle, and we proceed by induction from the leaves to the root of T.

Recall that the goal is to combine the descendant leaf bricks of a node u at level l in left-to-right order into one large brick of length at most $(1 + \lambda)^{l+1} v_l^{15}$. For the base case, if u is at level 1, the combining is done by prefix summation. Let N the number of input numbers falling

into the subinterval of $(0,1]$ associated with u. If $N \leq (1+\lambda)v_1^{15}$, then the total length of the bricks to be combined is no more than $(1+\lambda)^2 v_1^{15}$, i.e., u will not become bad. N is binomially distributed, and $E(N) = v_1^{15}$. Hence by a Chernoff bound, the probability that u becomes bad is at most $e^{-\lambda^2 v_1^{15}/3} = 2^{-\Omega(v_1^2)}$.

Now let u be a node at level $l \geq 2$ and of width v^{15}. We analyze in turn the three main reasons why u may become bad. We will assume n to be larger than some (unspecified) constant.

"Act of God": Call u *heavy* if some bucket represented by a leaf descendant of u contains more than v^2 elements. The expected number of elements in any fixed bucket being 1, a Chernoff bound shows the probability that any fixed bucket contains more than v^2 elements to be $2^{-\Omega(v^2)}$. Since u has only v^{15} leaf descendants, the probability that u is heavy is therefore $2^{-\Omega(v^2)}$.

"Bad Luck": Call the node u *well-supplied* if it is not heavy and has at most v^2 bad children. We show that a well-supplied node becomes bad with probability $2^{-\Omega(v^2)}$. Using Ragde compaction, u first attempts to place its bad children in an array of size v^{10}. If this fails, u clearly is not well-supplied and we give up. Otherwise, since (for sufficiently large values of n) even a bad child provides no more than $\frac{1}{2}v$ bricks (due to the rapid increase of $v_1, v_2, \ldots$, its width is no bigger), the compaction implicitly places descriptors of all bricks contributed by bad children in an array of size at most $\frac{1}{2}v^{11}$. We clearly cannot place descriptors of all bricks contributed by the good children in an array of comparable size, since there are close to v^{15} such arrays. At this point, however, we make a crucial observation. The bricks contributed by good children are all of the same length. Given a run of consecutive good children of u between two successive bad children, we can therefore combine the associated bricks into a single brick in a trivial way. Since there are at most (v^2+1) such runs, descriptors of combined bricks derived from the bricks provided by the good children of u can be compacted using Ragde compaction into an array of size $(v^2+1)^5$. Altogether, we have succeeded (for sufficiently large values of n) in placing descriptors of all bricks to be combined by u in an array of size v^{11}.

A brick provided by a child of u is either one of the original bricks $J_1, \ldots, J_n$ (that never participated in a successful combining), in which case its length is at most $(1+\lambda)v^2$ (since u is not heavy), or it is a brick created by some (good) nonleaf descendant u' of u at level $l' < l$, in which case its length is at most $(1+\lambda)^{l'+1} \leq (1+\lambda)^l$ times the width of u'. All bricks provided by good children of u are of the second kind. Hence the total length of the bricks to be combined by u is at most

$$(1+\lambda)v^2 \cdot v^{11} + (1+\lambda)^l v^{15} \leq (1+\lambda)^l v^{15}(1+1/v^2) \leq (1+\lambda)^l(1+\lambda/4)v^{15}$$

(recall that $v \geq v_2 = 2^{1/\lambda}$, so that $v^2 \geq 4/\lambda$). Run in constant time with v^{12} processors, the algorithm of Theorem 3.6 can therefore combine the at most v^{11} bricks into a brick of length at most $(1+2^{-\log\log v - 1})(1+\lambda)^l(1+\lambda/4)v^{15}$. Since $2^{-\log\log v-1} \leq \frac{1}{2\log v_2} = \lambda/2$ and $(1+\lambda/2)(1+\lambda/4) \leq 1+\lambda$, this is at most $(1+\lambda)^{l+1}v^{15}$, as desired.

If we actually run the algorithm v^2 times in parallel, the probability that all trials fail is $2^{-\Omega(v^2)}$. Altogether, the node u needs v^{14} processors. Since the number of nodes at the same level as u is n/v^{15}, the total processor requirements are n/v.

"Bad Subcontractors": Nodes at level $l-1$ are of width $(\log v)^{15}$. Hence by the inductive assumption, each node at level $l-1$ becomes bad with probability $2^{-\Omega((\log v)^2)} = v^{-\Omega(\log v)}$. Assume for a moment that the events of distinct nodes on the same level becoming bad were independent. Then the number of bad children of u would be bounded by a binomial distribution with expected value $v^{15} \cdot v^{-\Omega(\log v)} = o(1)$, and the probability of u having more than v^2 bad children would be $2^{-\Omega(v^2)}$, as desired, which finishes the analysis.

Nodes on the same level are not really independent, as assumed above. Since the dependencies are very weak and furthermore appear to work in our favor, this paragraph merely sketches the technical justification for what should already be plausible. What we do know is that the expected number of bad nodes at level $l-1$ is $n \cdot v^{-\Omega(\log v)}$. By a martingale argument, the actual number of such nodes is comparably small, except with negligible probability (we here gloss over some easy details). Furthermore, the set V of bad nodes at level $l-1$ is *symmetrically* distributed, by which we mean that $\Pr(V = V_1) = \Pr(V = V_2)$ whenever V_1 and V_2 are subsets of the set of all nodes at level $l-1$ with $|V_1| = |V_2|$. It now follows from Fact 5 of [19] that the probability that u has more than v^2 bad children is $2^{-\Omega(v^2)}$.

Finally note that the entire input is packed into a brick of length $(1+\lambda)^{O(\log^* n)}n$, which for sufficiently large values of n is bounded by $(1+t^{-t})n$. The combining at nodes at level 1 requires $O(\log v_1 / \log\log v_1) = O(t)$ time and $O(n)$ operations. The combining at nodes at level ≥ 2 takes constant time per level, a total of $O(\log^* n) = O(t)$ time, and never uses more than n/v_1 processors, so that the total number of operations executed is $O(n)$. ∎

6 Integer Sorting

In this section we consider the problem of (padded-)sorting a multiset of integers in the range $1..n$, both with and without the stability requirement. No assumption is made about the distribution of the input values. Note that for all $n, m \in \mathbb{N}$, (padded-)sorting a multiset of n integers in the range $1..m$ stably reduces to (padded-)sorting n distinct integers in the range $1..nm$.

The core of our algorithms for padded-sorting integers is a subroutine that approximately ranks n integers in the range $1..n^{\Theta(1)}$ using $(\log n)^{\Theta(1)}$ processors per input number. The top-level structure of this subroutine, described in Lemma 6.2, is similar to that of our comparison-based padded-sorting algorithm, but some new ideas (and the extra power of the OR PRAM) are needed because fast run times are desired with only a polylogarithmic number of processors per input number. The rest of the section describes how to reduce the size of the original problem by a polylogarithmic factor to make Lemma 6.2 applicable, both when stability is required and when it is not.

We begin by enumerating some tasks that can be done quickly on the OR PRAM for small inputs.

Lemma 6.1: For all integers $m, w \geq 4$ the following problems can be solved on an m-processor OR PRAM with $O(m)$ space and a word length of w bits, using tables that can be constructed in constant time using $2^{o(w)}$ processors and $2^{o(w)}$ space:

(a) Compute the prefix sums of m integers of $O(\log m)$ bits each in $O(\log m/\log w)$ time;

(b) Sort m integers of $O(\log m)$ bits each in $O((\log m/\log w)^2)$ time.

Proof: Let $r = \lceil w^{1/4} \rceil$ and $h = \log m/\log r = \Theta(\log m/\log w)$. If $\log m > r$, both claims are trivial, since then $h = \Omega(\log m/\log\log m)$. Assume hence that $\log m \leq r$.
(a) The prefix sums of r integers of $O(\log m)$ bits each can be computed in constant time by table lookup, and it is easy to see that the relevant table can be constructed as claimed in the lemma. Use this at every node of a tree with m leaves, degree at most r and height $O(h)$, with the input numbers fed into the leaves and the computation proceeding from the leaves to the root. Subsequently each prefix sum of the input numbers can be determined in $O(h)$ time by one processor from a root-to-leaf path in the tree.

(b) Since the algorithm of part (a) does not utilize the full word length, we can use it to carry out r simultaneous prefix summations. Assuming that the input consists of integers in the range $0..r-1$, use this to count the number of occurrences of each value in the input. By table lookup applied to the result (which is stored in $O(1)$ words), each input element can tell the number of smaller elements. Moreover, by inspecting the intermediate values computed at its ancestors in the tree, it can deduce the number of elements of the same value as itself to its left, i.e., it can compute its rank among the input elements. This gives a procedure for stably sorting integers in the range $0..r-1$ in $O(h)$ time. Use radix sort to sort the full input numbers of $O(\log m)$ bits each in a total time of $O(h^2)$. ∎

Lemma 6.2: There is a constant $\epsilon > 0$ such that for all given integers $n, k, t \geq 2$, n distinct integers in the range $1..n^2$ can be approximately ranked as follows with probability at least $1 - 2^{-n^\epsilon}$:

(a) With accuracy $2^{-t \log\log(kn)}$ on an OR PRAM using $O(t)$ time, $O(kn(\log n)^2)$ processors and $O(kn(\log n)^2)$ space;

(b) With accuracy t^{-t} on a standard PRAM using $O(t)$ time, $O(n(\log n)^2)$ processors and $O(n(\log n)^2)$ space, provided that $t \geq \log\log n$.

Proof: We again describe an algorithm with a larger failure probability, whose top-level structure is identical to that of Theorem 3.4. We give the proof for part (a) first and outline the modifications necessary for part (b) later.

Let $g = 48 \cdot 2^{t\lceil \log\log(kn) \rceil}$. If $g^{64} > n$, the input can be sorted using one of the algorithms of [12, 33]. Otherwise again let $h \geq 1$ be the largest integer with $g^{h-1} g^{64} \leq n$ and execute the following steps:

(1) Construct the tree T, i.e.,

(1.a) Compute the sets $S_0, \ldots, S_h$;

(1.b) Determine the parent of each node in $\bigcup_{i=1}^{h} S_i$;

(1.c) Sort each set of siblings in T;

(2) Determine the ancestors of each leaf in T;

(3) Compute the interval associated with each leaf.

For the implementation of Step (1.a), we essentially use the algorithm of Lemma 3.3 in parallel for $s = 1, g, \ldots, g^{h-1}$, which requires $O(hkn) = O(kn \log n)$ processors. Since the condition $t \geq \lceil \log n / \log k \rceil$ is not necessarily satisfied, we have to realize the steps of the algorithm slightly differently. For Step (1), simply use part (b) of Lemma 2.3 instead of part (a); as we have $\Omega(\log n)$ processors per element, the time needed is constant. For Step (2), use Lemma 6.1(a). Since the word length is at least $\Theta(\log(kn))$ bits, we obtain a run time of $O(\log g / \log\log(kn)) = O(t)$. Step (1.b) takes constant time using predecessor structures like those constructed in Step (1.a). For Step (1.c) we first padded-sort each set of siblings using the algorithm of Corollary 3.5, run $\Theta(\log n)$ times in parallel to achieve sufficient reliability, and then compact the resulting sequence by means of Lemma 6.1(a). In Step (2) each leaf v first computes its candidate predecessors $u_i = Pred(v, S_i)$ and $u_i' = Pred^+(u_i, S_i)$ at depth i, for $i = 0, \ldots, h$, which requires $O(hn \log n) = O(n(\log n)^2)$ processors. The entire subgraph of the tree T induced by $\{u_0, u_0', \ldots, u_h, u_h'\}$ can now be represented in $O(h) = O(\log n)$ bits by noting for each of u_{i+1} and u_{i+1}', for $i = 0, \ldots, h-1$, whether its parent is u_i, u_i' or neither of these, after which the ancestors of v can be determined in constant time using table lookup. Recalling that the compounded interval of each leaf is determined by $O(\log n)$ bits, it is easy to see that Step (3) can be executed in constant time in a similar way.

This proves the lemma for the OR PRAM. For a standard PRAM we take $g = 48 \cdot t^t$, employ usual prefix summation instead of Lemma 6.1(a) and carry out Steps (2) and (3) using standard pointer-doubling techniques. ∎

Theorem 6.3: There is a constant $\epsilon > 0$ such that for all given integers $n, k \geq 4$ and $t \geq \lceil \log\log n / \log k \rceil$, n integers in the range $1..n$ can be stably approximately ranked with accuracy $2^{-t\log\log(kn)}$ on an OR PRAM using $O(t)$ time, kn processors and $O(kn)$ space with probability at least $1 - 2^{-n^\epsilon}$. For $t \geq \log\log n$ the result holds for a standard PRAM with ranking accuracy t^{-t}.

Proof: We give the proof for the OR PRAM. As usual, we describe a simpler algorithm with a somewhat larger failure probability. Let X be the input set, take $g = \min\{2^{t\lceil\log\log(kn)\rceil}, n+1\}$ and let $\lambda = 1/(4g)$. The algorithm consists of the following main steps:

(A) Compute a sample C of X of size $O(n/(\log n)^2)$ with $maxgap(C, X) = g^{O(1)}$ that includes the first element of X and has spread at most λ in X. For $i = 1, \ldots, |C|$, let C_i be the ith segment of C in X;

(B) Rank C approximately with accuracy λ using the algorithm of Lemma 6.2. We can assume that this marks C_i with a label q_i, for $i = 1, \ldots, |C|$, such that with $q_0 = 0$, we have $\frac{1}{|C|} \leq q_i - q_{i-1} \leq \frac{1+\lambda}{|C|}$, for $i = 1, \ldots, |C|$;

(C) For $i = 1, \ldots, |C|$, sort C_i (without padding);

(D) Output the sequence of labels of the elements of X computed as follows: If $x \in X$ is the lth element of C_j, then the label of x is $(1+\lambda)(q_{j-1} + \frac{l}{|C_j|}(q_j - q_{j-1}))$.

For $i = 1, \ldots, n$, let r_i be the label computed in Step (D) for the element in X of rank i. It is easy to see that for each $i \in \{1, \ldots, n\}$, there is a $j \in \{1, \ldots, |C|\}$ such that

$$\frac{1}{n} \leq \frac{1+\lambda}{|C| maxgap(C,X)} \leq (1+\lambda)\frac{q_j - q_{j-1}}{|C_j|} = r_i - r_{i-1} \leq \frac{(1+\lambda)^2}{|C| mingap(C,X)} \leq \frac{(1+\lambda)^3}{n} \leq \frac{1+1/g}{n}.$$

This demonstrates the correctness of the algorithm. We now discuss the implementation of Steps (A) and (C).

Step (A) is carried out using the algorithm of Lemma 3.3, modified as described for Step (1.a) in the proof of Lemma 6.2. Since we no longer have $\Omega(\log n)$ processors per element, the time bound becomes $O(\lceil \log\log n / \log k \rceil)$. In Step (C) we break each segment C_i into pieces of polylogarithmic size, each of which can be sorted in $O(\lceil \log\log n / \log k \rceil)$ time, and then concatenate the sorted pieces. In more detail, Step (C) consists of the following substeps:

(C.1) Compute a sample S of X with $|S| = O(n/(\log n)^2)$ and $maxgap(S, X) = O((\log n)^3)$;

(C.2) Padded-sort $C \cup S$ with padding factor $O(1)$ using Lemmas 6.2 and 3.1;

(C.3) Partition X according to $C \cup S$;

(C.4) Padded-sort each segment of X induced by $C \cup S$ with padding factor $O(1)$;

(C.5) For $i = 1, \ldots, |C|$, complete the sorting of C_i.

Step (C.1) is easy, since taking S as a naive sample of X of size $\Theta(n/(\log n)^2)$ will do. Steps (C.3) and (C.4) are carried out using Lemma 2.3(b) and Corollary 3.5, respectively. The task in Step (C.5) is, for $i = 1, \ldots, |C|$, to compact the subsegments of C_i sorted in Step (C.4) and to place them next to each other in the right order. What this involves is to use Lemma 6.1(a) to compute prefix sums, first within each segment of X induced by $C \cup S$, and afterwards within each segment of S induced by C (as computed in Step (C.2)). With high probability, this takes $O(\log g / \log\log(kn)) = O(t)$ time. ∎

Theorem 6.4: There is a constant $\epsilon > 0$ such that for all given integers $n, t \geq 2$, n integers in the range $1..n$ can be (unstably) approximately ranked with accuracy $2^{-t \log\log n}$ on an OR PRAM using $O(t)$ time, $O(n)$ operations and $O(n)$ space with probability at least $1 - 2^{-n^\epsilon}$. For $t \geq \log\log n$ the result holds for a standard PRAM with ranking accuracy t^{-t}.

Proof: Again we give the proof for the OR PRAM. Although any nondegenerate input contains duplicate keys, for the purpose of sampling we have to consider all keys to be distinct. We begin by semisorting the input keys by their values into an array Q and define a particularly convenient total order among the keys of a common value as that given by their order in Q. It is now easy to see that for any nonempty subset S of the input set X, a predecessor structure for S that can be queried in constant time by one processor for arguments in X can be constructed in constant time with n processors: For the preprocessing, apply Lemma 2.3(c) both to the set of positions in Q that contain elements of S and to the set of values that occur in S, and determine for each value occurring in S the largest (i.e., rightmost in Q) key of that value. The predecessor of an element $x \in X$ is now found as the nearest element of S preceding it in Q, unless the value of that element is different from that of x, in which case the predecessor of x is the largest element in S of the preceding value. If we substitute this predecessor structure for the one used in the proof of Theorem 6.3 and take $g = \min\{2^{t\lceil \log\log n \rceil}, n+1\}$, only Step (C.4) needs to be modified.

Recall that Step (C.4) sorts a collection of segments of polylogarithmic sizes. If subtracting an arbitrary value occurring in a segment from all keys in the segment leaves keys of absolute value bounded by $(\log n)^4$, the segment can be sorted in constant time using Lemma 6.1(b). On the other hand, the number of segments violating this condition is $O(n/(\log n)^4)$, so that we can allocate $\Theta(\log n)$ processors to each key in each such segment and padded-sort the segment in constant time using Corollary 3.5. ∎

7 Approximate Prefix Summation

Recall that the approximate prefix summation problem with accuracy λ is, given n nonnegative integers $x_1, \ldots, x_n$, to compute n integers $y_1, \ldots, y_n$ such that with $y_0 = 0$, the following holds for $i = 1, \ldots, n$:

(1) $y_i \geq y_{i-1} + x_i$;
(2) $y_i \leq (1+\lambda) \sum_{j=1}^{i} x_j$.

Theorem 7.1: There is a constant $\epsilon > 0$ such that for all given integers $n, t \geq 4$, approximate prefix summation problems of size n and with input numbers of size polynomial in n can be solved with accuracy $2^{-t \log\log n}$ on an OR PRAM using $O(t)$ time, $O(n)$ operations and $O(n)$ space with probability at least $1 - 2^{-n^\epsilon}$.

Proof: Let the input be $x_1, \ldots, x_n$ and take $s_i = \sum_{j=1}^{i} x_j$, for $i = 1, \ldots, n$, and $\lambda = 2^{-t\lceil \log\log n \rceil}$. Consider first the simple *unary* case where $x_i \in \{0, 1\}$, for $i = 1, \ldots, n$. Derive from the input a new sequence $z_1, \ldots, z_n$ by setting $z_i = i$ if $x_i = 1$ and $z_i = n+1$ if $x_i = 0$, for $i = 1, \ldots, n$. Then padded-sort $z_1, \ldots, z_n$ with padding factor λ using the algorithm of Theorem 6.4 combined with Lemma 3.1 and note that for $i = 1, \ldots, n$, if $x_i = 1$, then s_i is precisely the rank of z_i in the resulting sorted sequence. This quantity, of course, is not readily available, but it is approximated well by the position of z_i in the output array, which we therefore take to be y_i. This defines y_i for all $i \in \{1, \ldots, n\}$ with $x_i = 1$. Compute y_i for all other i by "copying the nearest value to the left", i.e., by letting $y_i = y_j$, where $j < i$ is maximal with $x_j = 1$ ($y_i = 0$ if no such j exists). Using Lemma 2.3(c), this can be done in constant time.

Condition (1) in the definition of approximate prefix summation is easily seen to be satisfied. As for condition (2), since the padded sorting of $z_1, \ldots, z_n$ places the element of rank i in position at most $(1+\lambda)i$ of the output array, for $i = 1, \ldots, n$, we have $y_i \leq (1+\lambda)s_i$, for $i = 1, \ldots, n$.

We now move to the general case of input numbers drawn from a set $U = \{0, \ldots, M\}$, where $M = n^{O(1)}$, and first describe a preprocessing phase that reduces the problem size by $\Theta(\log n)$. Since this is easy if $t = \Omega(\log\log n)$, we can assume that $\lambda \geq 2^{-(\log\log n)^2}$. Without loss of generality suppose that the sum of the input numbers also belongs to U. The basic idea is to proceed as in the proof of Lemma 6.1(a), i.e., to use a tree of constant height and degree $\Theta((\log n)^{1/2})$ to compute prefix sums within groups of $\Theta(\log n)$ input numbers, the prefix summation at each node being performed in constant time by table lookup. The immediate difficulty with this approach is that the input numbers are of $\Theta(\log n)$ bits each, so that $\Theta((\log n)^{1/2})$ of them will not fit into one word. We therefore represent the values involved only approximately, using fewer bits. More precisely, for every fixed γ, there is a set $R \subseteq U$ of "representable" numbers such that

(1) The elements of R can be represented using $(\log\log n)^{O(1)}$ bits, and the corresponding encoding and decoding functions can be evaluated in constant time by a single processor;

(2) For every $x \in U$, there is a "rounded value" $r(x) \in R$ with $x \leq r(x) \leq (1+\mu)x$, where $\mu = 2^{-(\log\log n)^{\gamma}}$. Moreover, $r(x)$ can be computed in constant time by a single processor.

Given $x, y \in R$, let $x \oplus y = r(\min\{x+y, M\})$ and note that $\oplus$ can be evaluated in constant time by a single processor. $\oplus$ serves as our approximation to addition. Every time it is applied, we incur a relative error of up to μ; if γ is chosen sufficiently large, however, no computed quantity will be larger than $1 + \frac{\lambda}{4}$ times its "true value". Now n processors can in constant time construct a table that "takes as input" $\Theta((\log n)^{1/2})$ numbers $x_1, x_2, x_3, \ldots$ in R and "produces as output" their approximate prefix sums $x_1, x_1 \oplus x_2, (x_1 \oplus x_2) \oplus x_3$, etc. Using this table it is easy to realize the basic idea for the preprocessing described above.

In order to compute approximate prefix sums for input numbers $x_1, \ldots, x_n$ of $O(\log n)$ bits each, we apply the algorithm for the unary case in parallel to every bit position of the input numbers (this is where we need a superlinear number of processors) with a required accuracy of $1 + \frac{\lambda}{4}$. This yields each approximate prefix sum as a collection of $\Theta(\log n)$ values, one for each bit position, that can be added with relative error at most $\frac{\lambda}{4}$ as in the preprocessing. Even though the input numbers $x_1, \ldots, x_n$ may also be affected by relative errors of up to $\frac{\lambda}{4}$ due to the preprocessing, the overall relative error in the output variables will be at most $(1+\frac{\lambda}{4})^3 - 1 \leq \lambda$. ∎

Replacing the constant-time preprocessing described above by the straightforward prefix summation in groups of size $\Theta(\log n)$, it is easy to obtain a result for the standard PRAM identical to that of Theorem 7.1, except that we must require $t \geq \log\log n$, and that the accuracy obtained is t^{-t}; below we refer to the corresponding algorithm as the *basic algorithm*. We can do better, however. Assuming only that standard operations like addition take constant time on integers as large as the input numbers, we can allow the latter to be any nonnegative integers whatsoever, rather than integers of $O(\log n)$ bits.

Theorem 7.2: There is a constant $\epsilon > 0$ such that for all given integers $n \geq 4$ and $t \geq \log\log n$, approximate prefix summation problems of size n can be solved with accuracy t^{-t} using $O(t)$ time, $O(n)$ operations and $O(n)$ space with probability at least $1 - 2^{-n^{\epsilon}}$.

Proof: We describe an algorithm with a larger failure probability. Since for $t \geq \log n/\log\log n$ an accuracy of 2^{-2^t} can be obtained through simple rounding to $2^{\Theta(t)}$ bits followed by exact

prefix summation [22], we will assume that $\lambda \geq 12/n$, where $\lambda = t^{-t}$.

The basic idea is that if the input numbers vary widely in size, then the smallest of them can be safely ignored. This is not quite true, since a small number that is preceded only by other small numbers cannot necessarily be ignored, but the observation enables us to divide the sequence of input numbers into segments, within each of which numbers can be rounded to $O(\log n)$ bits.

In more detail, let the input be $x_1, \ldots, x_n$ and begin by computing the prefix maxima $m_1, \ldots, m_n$, where $m_i = \max(\{x_j : 1 \leq j \leq i\} \cup \{1\})$, for $i = 1, \ldots, n$. This is easy to do in $O(\log \log n)$ time; we omit the details. The next task is to partition the index set $\{1, \ldots, n\}$ into consecutive segments $S_1, \ldots, S_v$. We define the division into segments by describing the set of indices $i \in \{1, \ldots, n-1\}$ such that i and $i+1$ are *separated*, i.e., belong to distinct segments. In fact, we operate with *strong* and *weak* separations. For $i = 1, \ldots, n-1$, i is strongly separated from $i+1$ exactly if $m_{i+1} \geq n^2 \cdot m_i$. Intuitively, a strong separation implies a splitting into practically independent subproblems: The numbers in the left subproblem add up to less than $1/n$ times the first number in the right subproblem and can therefore essentially be ignored in the solution of the right subproblem.

Assume that $S = \{i, \ldots, j\}$ is a (maximal) segment remaining after the introduction of strong separations. We describe how S is split into a collection of final segments by means of weak separations. Ideally, we would like to split S at the prefix maxima $m_i \cdot n^2, m_i \cdot n^4, m_i \cdot n^6$, etc., i.e., to separate two indices l and $l+1$ exactly if for some integer $h \geq 1$, $m_l < m_i \cdot n^{2h}$, but $m_{l+1} \geq m_i \cdot n^{2h}$. Not knowing how to do this efficiently using only standard operations, we proceed as follows: Associate a processor with each element of S and let these processors generate the sequence $m_i \cdot \overline{n}^2, m_i \cdot \overline{n}^4, \ldots, m_i \cdot \overline{n}^{2(j-i+1)}$, where $\overline{n}$ is the smallest power of 2 no smaller than n ($\overline{n}$ is substituted for n because it is easy to compute powers of $\overline{n}$). Then stably merge this sequence with the sequence $m_i, \ldots, m_j$ of prefix maxima and (weakly) separate l and $l+1$, for $l = i, \ldots, j-1$, exactly if m_l and m_{l+1} are not consecutive in the resulting sequence (i.e., if they are separated by an element $m_i \cdot \overline{n}^{2h}$ of the first sequence).

Let $\mathcal{S} = \{S_1, \ldots, S_v\}$ be the sequence of segments resulting from the execution of the above procedure. For each segment $S = \{i, \ldots, j\} \in \mathcal{S}$, if each number in S is rounded to the nearest larger multiple of $\max\{\lfloor m_i/n^2 \rfloor, 1\}$, we commit a relative error of at most $1/n$, which is negligible (we ignore it in the following). Furthermore, by construction of the segments, no number in S is larger than $m_i \cdot \overline{n}^2$, so that the rounding expresses each element in S in $O(\log n)$ bits. This means that the basic algorithm can be applied to (the sequence of numbers with indices in) S. The same holds for the union of two consecutive segments, provided that they are only weakly separated. We now compute preliminary prefix sums $\tilde{y}_1, \ldots, \tilde{y}_n$ as follows: Let $i \in \{1, \ldots, n\}$ and suppose that $i \in S_j$. Then $\tilde{y}_i$ is computed by applying the basic algorithm $\Theta(\log n)$ times in parallel to S_j if $j = 1$ or S_{j-1} and S_j are strongly separated, and to $S_{j-1} \cup S_j$ if $j \geq 2$ and S_{j-1} and S_j are only weakly separated. For these applications of the basic algorithm we will require an accuracy of $\lambda/5$. Since each segment participates in at most two applications of the basic algorithm, the computation of $\tilde{y}_1, \ldots, \tilde{y}_n$ uses $O(n)$ operations. For $j = 1, \ldots, v$, let $q_j = \tilde{y}_{\min S_j}$. For $i \in S_1$, the final prefix sum y_i is taken to be simply $\tilde{y}_i$. For $i \in S_j$ with $j \geq 2$, let $y_i = \tilde{y}_i + \lfloor \frac{\lambda}{4}(q_j + q_{j-1}) \rfloor$.

We must show that the algorithm is correct, i.e., that $y_1, \ldots, y_n$ satisfy conditions (1) and (2) for $i = 1, \ldots, n$. We leave the (easy) case $i = 1$ to the reader; hence let $i \in \{2, \ldots, n\}$. If $i-1$ and i are not separated, the condition $y_i \geq y_{i-1} + x_i$ is equivalent to $\tilde{y}_i \geq \tilde{y}_{i-1} + x_i$, and it is satisfied by the correctness of the basic algorithm and the fact that the rounding within segments is upwards. Suppose now that $i-1$ and i are separated, say $i-1 \in S_{j-1}$ and $i \in S_j$. The

condition $y_i \geq y_{i-1} + x_i$ is then satisfied if $(1+\frac{\lambda}{4})\widetilde{y}_i \geq \widetilde{y}_{i-1} + x_i + \frac{\lambda}{4}q_{j-2} + 1$, where $q_0 = 0$. Since $q_{j-2} \leq (1+\frac{\lambda}{5})\frac{x_i}{n}$ and $x_i \geq n^2$, it suffices to prove that $(1+\frac{\lambda}{4})\widetilde{y}_i \geq \widetilde{y}_{i-1} + (1+\frac{1}{n})x_i$. If $i-1$ and i are strongly separated, this is easy, since $\widetilde{y}_i \geq x_i$ and $\widetilde{y}_{i-1} \leq (1+\frac{\lambda}{5})s_{i-1} \leq (1+\frac{\lambda}{5})\frac{x_i}{n}$. If $i-1$ and i are not strongly separated, let $w_h = \sum_{l \in S_h} x_l$, for $h \in \{j-1, j-2\}$ (take $w_0 = 0$) and observe that $\widetilde{y}_i \geq w_{j-1} + x_i$ and $\widetilde{y}_{i-1} \leq (1+\frac{\lambda}{5})(w_{j-1} + w_{j-2}) \leq (1+\frac{\lambda}{5})w_{j-1} + (1+\frac{\lambda}{5})\frac{x_i}{n}$, from which the desired relation follows.

Condition (2) is also satisfied, since for $i \in S_j$ with $j \geq 2$ we have $y_i = \widetilde{y}_i + \lfloor \frac{\lambda}{4}(q_j + q_{j-1}) \rfloor \leq (1+\frac{\lambda}{5})s_i + \frac{\lambda}{2}(1+\frac{\lambda}{5})s_i \leq (1+\lambda)s_i$. ∎

References

[1] M. Ajtai, J. Komlós, and E. Szemerédi, An $O(n \log n)$ sorting network, *in* Proc. 15th STOC (1983), pp. 1–9.

[2] M. Ajtai and M. Ben-Or, A theorem on probabilistic constant depth computations, *in* Proc. 16th STOC (1984), pp. 471–474.

[3] N. Alon and Y. Azar, The average complexity of deterministic and randomized parallel comparison-sorting algorithms, *SIAM J. Comput.* **17** (1988), pp. 1178–1192.

[4] Y. Azar and U. Vishkin, Tight comparison bounds on the complexity of parallel sorting, *SIAM J. Comput.* **16** (1987), pp. 458–464.

[5] H. Bast and T. Hagerup, Fast and reliable parallel hashing (preliminary version), *in* Proc. 3rd SPAA (1991), pp. 50–61.

[6] H. Bast, M. Dietzfelbinger, and T. Hagerup, A perfect parallel dictionary, *in* Proc. 17th MFCS (1992), LNCS 629, pp. 133–141.

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

[8] O. Berkman and U. Vishkin, Recursive *-tree parallel data-structure, *in* Proc. 30th FOCS (1989), pp. 196–202.

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

[10] R. B. Boppana, The average-case parallel complexity of sorting, *IPL* **33** (1989), pp. 145–146.

[11] B. S. Chlebus, Parallel iterated bucket sort, *IPL* **31** (1989), pp. 181–183.

[12] R. Cole, Parallel merge sort, *SIAM J. Comput.* **17** (1988), pp. 770–785.

[13] R. Cole and U. Vishkin, Faster optimal parallel prefix sums and list ranking, *Inform. and Comp.* **81** (1989), pp. 334–352.

[14] L. Devroye, *Lecture Notes on Bucket Algorithms*, Birkhäuser, Boston, MA, 1986.

[15] J. Gil, Y. Matias, and U. Vishkin, Towards a theory of nearly constant time parallel algorithms, *in* Proc. 32nd FOCS (1991), pp. 698–710.

[16] M. T. Goodrich, Using approximation algorithms to design parallel algorithms that may ignore processor allocation, *in* Proc. 32nd FOCS (1991), pp. 711–722.

[17] T. Hagerup, Towards optimal parallel bucket sorting, *Inform. and Comp.* **75** (1987), pp. 39–51.

[18] T. Hagerup, Hybridsort revisited and parallelized, *IPL* **32** (1989), pp. 35–39.

[19] T. Hagerup, Constant-time parallel integer sorting, *in* Proc. 23rd STOC (1991), pp. 299–306.

[20] T. Hagerup, Fast parallel space allocation, estimation and integer sorting, TR MPI–I–91–106, MPI für Informatik, 6600 Saarbrücken, Germany, 1991. Preliminary version *in* Proc. 23rd STOC (1991).

[21] T. Hagerup, The log-star revolution, *in* Proc. 9th STACS (1992), LNCS 577, pp. 259–278.

[22] T. Hagerup, The parallel complexity of integer prefix summation, TR LSI–92–18–R, Dept. de LSI, Universitat Politècnica de Catalunya, 08028 Barcelona, Spain, 1992.

[23] J. Håstad, Almost optimal lower bounds for small depth circuits, *in* Proc. 18th STOC (1986), pp. 6–20.

[24] J. Håstad, Personal communication, Sep. 1992.

[25] A. Itai, A. G. Konheim, and M. Rodeh, A sparse table implementation of priority queues, *in* Proc. 8th ICALP (1981), LNCS 115, pp. 417–431.

[26] P. D. MacKenzie and Q. F. Stout, Ultra-fast expected time parallel algorithms, *in* Proc. 2nd SODA (1991), pp. 414–423.

[27] P. D. MacKenzie and Q. F. Stout, Ultra-fast expected time parallel algorithms, TR CSE–TR–115–91, Dept. of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, MI 48109, USA, 1991.

[28] P. D. MacKenzie, Load balancing requires $\Omega(\log^* n)$ expected time, *in* Proc. 3rd SODA (1992), pp. 94–99.

[29] Y. Matias and U. Vishkin, On parallel hashing and integer sorting, *J. Algorithms* **12** (1991), pp. 573–606.

[30] Y. Matias and U. Vishkin, Converting high probability into nearly-constant time — with applications to parallel hashing, *in* Proc. 23rd STOC (1991), pp. 307–316.

[31] M. D. McLaren, Internal sorting by radix plus sifting, *J. ACM* **13** (1966), pp. 401–411.

[32] P. Ragde, The parallel simplicity of compaction and chaining, *in* Proc. 17th ICALP (1990), LNCS 443, pp. 744–751.

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

[34] R. Raman, The power of Collision: Randomized parallel algorithms for chaining and integer sorting, *in* Proc. 10th FST & TCS Conf. (1990), LNCS 472, pp. 161–175. Also as TR 336, Dept. of Computer Science, University of Rochester, Rochester, NY 14627, USA, 1991.

[35] R. Raman, Optimal sub-logarithmic time integer sorting on the CRCW PRAM (note), TR 370, Dept. of Computer Science, University of Rochester, Rochester, NY 14627, USA, 1991.

[36] J. H. Reif and L. G. Valiant, A logarithmic time sort for linear size networks, *J. ACM* **34** (1987), pp. 60–76.

[37] R. Sarnath, Very fast parallel comparison sorting, Manuscript, 1991.

[38] R. Sarnath, Lower bounds for padded sorting and approximate counting, Manuscript, 1992.

[39] D. E. Willard, A density control algorithm for doing insertions and deletions in a sequentially ordered file in a good worst-case time, *Inform. and Comp.* **97** (1992), pp. 150–204.