PREPRINT. Journal of Computer and System Sciences, vol 57, pp 74-93, 1998.

Sorting in Linear Time?*

Arne Andersson† Torben Hagerup‡
Stefan Nilsson§ Rajeev Raman¶

*A preliminary version of this paper was presented at the 27th Annual ACM Symposium on the Theory of Computing in Las Vegas, Nevada in May 1995.

†Department of Computer Science, Lund University, Box 118, S–221 00 Lund, Sweden. arne@dna.lth.se

‡Max-Planck-Institut für Informatik, D–66123 Saarbrücken, Germany. Supported by the ESPRIT Basic Research Actions Program of the EU under contract No. 7141 (project ALCOM II). torben@mpi-sb.mpg.de

§Helsinki University of Technology, Department of Computer Science, Otakaari 1, 02150 Espoo, Finland. The work was done while this author was at the Department of Computer Science, Lund University, Sweden. sni@cs.hut.fi

¶Department of Computer Science, King's College London, Strand, London WC2R 2LS, U. K. raman@dcs.kcl.ac.uk

Proposed running head: Sorting in Linear Time?

Contact author: Torben Hagerup, Max-Planck-Institut für Informatik, D–66123 Saarbrücken, Germany. `torben@mpi-sb.mpg.de`

Abstract

We show that a unit-cost RAM with a word length of w bits can sort n integers in the range $0\,..\,2^w - 1$ in $O(n \log\log n)$ time, for arbitrary $w \geq \log n$, a significant improvement over the bound of $O(n\sqrt{\log n})$ achieved by the fusion trees of Fredman and Willard. Provided that $w \geq (\log n)^{2+\epsilon}$ for some fixed $\epsilon > 0$, the sorting can even be accomplished in linear expected time with a randomized algorithm.

Both of our algorithms parallelize without loss on a unit-cost PRAM with a word length of w bits. The first one yields an algorithm that uses $O(\log n)$ time and $O(n \log\log n)$ operations on a deterministic CRCW PRAM. The second one yields an algorithm that uses $O(\log n)$ expected time and $O(n)$ expected operations on a randomized EREW PRAM, provided that $w \geq (\log n)^{2+\epsilon}$ for some fixed $\epsilon > 0$.

Our deterministic and randomized sequential and parallel algorithms generalize to the lexicographic sorting problem of sorting multiple-precision integers represented in several words.

1 Introduction

Sorting is one of the most fundamental computational problems, and n keys can be sorted in $O(n \log n)$ time by any of a number of well-known sorting algorithms. These algorithms operate in the *comparison-based* setting, i.e., they obtain information about the relative order of keys exclusively through pairwise comparisons. It is easy to show that a running time of $\Theta(n \log n)$ is optimal in the comparison-based model. However, this model may not always be the most natural one for the study of sorting problems, since real machines allow many other operations besides comparison. Using indirect addressing, for instance, it is possible to sort n integers in the range $0 \,..\, n-1$ in linear time via bucket sorting, thereby demonstrating that the comparison-based lower bound can be meaningless in the context of integer sorting.

Integer sorting is not an exotic special case, but in fact is one of the sorting problems most frequently encountered. Aside from the ubiquity of integers in algorithms of all kinds, we note that all objects manipulated by a conventional computer are represented internally by bit patterns that can be interpreted as integers by the built-in arithmetic instructions. For most basic data types, the numerical ordering of the representing integers induces a natural ordering on the objects represented; e.g., if an integer represents a character string in the natural way, the induced ordering is the lexicographic ordering among character strings. This is true even for floating-point numbers; indeed, the IEEE 754 floating-point standard was designed specifically to facilitate the sorting of floating-point numbers by means of integer-sorting subroutines [25, p. 228]. Many sorting problems therefore eventually boil down to sorting integers or, possibly, multiple-precision integers stored in several words.

Classical algorithms for integer sorting require assumptions about the size of the integers to be sorted, or else have a running time dependent on the size. In order to work in linear time, bucket sorting requires its input keys to come from a range of linear size. Radix sorting in k phases, each phase implemented via bucket sorting, can sort n integers in the range $0 \,..\, n^k - 1$ in $O(nk)$ time. A more sophisticated technique, due to Kirkpatrick and Reisch [26], reduces this to $O(n \log k)$, for arbitrary $k \geq 2$, but the fact remains that as the size of the integers to be sorted grows to infinity, the cost of the sorting also grows to infinity (or to $\Theta(n \log n)$, if we switch to a comparison-based method at the appropriate point).

If we allow intermediate results containing many more bits than the input numbers, while maintaining a unit-cost assumption, we can actually sort integers in linear time independently of their size, as demonstrated by Paul and Simon [31] and Kirkpatrick and Reisch [26]. But again, from a practical point of view, this is not what we want, since a real machine is unlikely to have unit-time instructions for operating on integers containing a huge number of bits. Instead, if the input numbers are w-bit integers, we would like all intermediate results computed by a sorting algorithm to fit in w bits as well—in the terminology of Kirkpatrick and Reisch, the algorithm should be *conservative.* In this case the assumption of a full repertoire of "reasonable" constant-time instructions is realistic. In the remainder of the paper, when nothing else is stated, we will take "sorting" to mean sorting w-bit words on a unit-cost RAM with a word length of w bits.

Fredman and Willard [18] were the first to show that n arbitrary integers can be sorted in $o(n \log n)$ time by a conservative method. Their fastest algorithm, a direct application of their *fusion-tree* data structure, sorts n integers in $O(n\sqrt{\log n})$ time. We describe two simple faster conservative sorting algorithms.

Our first algorithm works in $O(n \log \log n)$ time. It uses arithmetic instructions drawn from what we call the *restricted instruction set*, including comparison, addition, subtraction, bitwise AND and OR, and unrestricted bit shift, i.e., shift of an entire word (with zero filling) by a number of bit positions specified in a second word. As is not difficult to see, these instructions are all in AC^0, i.e., they can be implemented through constant-depth, polynomial-size circuits with unbounded fan-in. Since this is known not to be the case for the multiplication instruction [7], which is essential for the fusion-tree algorithm, our algorithm can also be viewed as placing less severe demands on the underlying hardware; this answers a question posed by Fredman and Willard (an answer to this question is already implicit in [5]). Also, the algorithm by Fredman and Willard is *nonuniform*, in the sense that a number of precomputed constants depending on w need to be included in the algorithm. Our algorithms need to know the value of w itself, but use no other precomputed constants.

Our second algorithm is randomized and works in $O(n)$ expected time, provided that $w \geq (\log n)^{2+\epsilon}$ for some fixed $\epsilon > 0$. Sufficiently large integers can thus be sorted in linear expected time by a conservative algorithm. The algorithm uses a *full instruction set* that augments the restricted instruction set with instructions for multiplication and random choice, where the latter takes an operand s in the range $1 \,..\, 2^w - 1$ and returns a random integer drawn from the uniform distribution over $\{1, \ldots, s\}$ and independent of all other such integers.

A difference in emphasis between the present paper and that of Fredman and Willard [18] should be noted. Whereas our concern is the fundamental problem of sorting conservatively as fast as possible, perhaps using superlinear space or randomization, Fredman and Willard chose to focus on the problem of deterministic conservative sorting in linear space, and their main algorithm solves this problem in $O(n \log n / \log \log n)$ time (for recent improvements of the latter result, see [4, 33]). Our first algorithm uses $O(n + 2^{w/r})$ space, for arbitrary fixed $r \geq 1$. Allowing randomization and the full instruction set, we can reduce this to $O(n)$ by means of universal hashing, while keeping an expected running time of $O(n \log \log n)$. Our second algorithm works in $O(n)$ space, even without the use of hashing.

Ben-Amram and Galil [8, Theorem 5] have shown that, in some circumstances, sorting requires $\Omega(n \log n)$ time on a RAM with an instruction set consisting of comparison, addition, subtraction, multiplication, and bitwise boolean operations. While it is possible to simulate *left* shifts using multiplication in their model, their lower bound does not apply if *right* shifts are allowed. We, on the other hand, assume that the complexity of left and right shifts is the same (as indeed it is to the underlying hardware).

Our basic algorithms can be extended in various directions. They parallelize without loss on a PRAM with a word length of w bits, yielding algorithms that use $O(\log n)$ time and $O(n \log \log n)$ operations on a deterministic CRCW PRAM or, provided that $w \geq (\log n)^{2+\epsilon}$ for some fixed $\epsilon > 0$, $O(\log n)$ expected time and $O(n)$ expected operations on a randomized EREW PRAM. The most comparable previous results are $O(\log n / \log \log n + \log w)$ time and $O(n \log w)$ operations on a deterministic CRCW PRAM [9], and $O(\log n \log \log n)$ expected time and $O(n\sqrt{\log n})$ expected operations on a randomized CREW PRAM [2]. The variant of the CRCW PRAM intended here and throughout the paper is the ARBITRARY PRAM, on which some participating processor wins and writes its value in the case of a write conflict.

We also obtain sequential and parallel algorithms for the general *lexicographic* sorting problem of sorting variable-length multiple-precision integers. As an example, if the n input numbers occupy a total of N words, they can be sorted sequentially in $O(N + n \log \log n)$ time; this is worst-case optimal if $N = \Omega(n \log \log n)$, since $\Omega(N)$ operations are needed just to scan the input.

Our results flow from the combination of the two techniques of *packed sorting* and *range reduction*. Packed sorting, introduced by Paul and Simon [31] and developed further in [24] and [2], saves on integer sorting by packing several integers into a single word and operating simultaneously on all of them at unit cost. This is only possible, of course, if several integers to be sorted fit in one word, i.e., packed sorting is inherently nonconservative. Range reduction, on the other hand, which underlies both radix sorting and the algorithm of Kirkpatrick and Reisch [26], reduces the problem of sorting integers in a certain range to that of sorting integers in a smaller range. The combination of the two techniques is straightforward: First range reduction is applied to replace the original full-size integers by smaller integers of which several fit in one word, and then these are sorted by means of packed sorting.

Our results use existing range reductions and packed-sorting algorithms as well as new range reductions and packed-sorting algorithms developed here. In Section 2 we combine an existing range reduction with an existing packed-sorting algorithm to obtain our deterministic sequential result. In Section 3 we introduce a new range reduction based on the use of *signatures*, short unique identifiers for long bit strings; the resulting sorting algorithm is called *signature sort*. Sections 4 and 5 contain remarks on extensions to sorting multiple-precision integers and the space requirements of our sorting algorithms, respectively. Sections 6–9 deal with parallel sorting. In Section 6 we describe a simple parallel packed-sorting algorithm and use it to obtain work-optimal parallelizations of the result of Section 2 on the CRCW PRAM. In Section 7 we develop a more refined version of this parallel packed-sorting algorithm and use it to obtain a time-optimal and work-optimal parallelization of signature sort on the EREW PRAM. Sections 8 and 9 contain parallel algorithms for sorting multiple-precision integers and some remarks on the space requirements of the parallel algorithms, respectively.

Given a set of integers to be sorted, we distinguish between *value-sorting*, whose output is a sorted array containing the same multiset of integers as the input, and *rank-sorting*, where each input key is to be labeled with its rank (with ties between equal elements resolved arbitrarily). A rank-sorting algorithm can always be used in place of a value-sorting algorithm—given the ranks, it is trivial to create a sorted output array—but the converse may not be true. For instance, rank-sorting can be used as above to sort a set of records according to some integer key, whereas it is not clear how to accomplish this by value-sorting the keys. When nothing else is stated, we will follow common practice and take "sorting" to mean rank-sorting, and our main results all pertain to rank-sorting. The relevance of value-sorting to the present paper is that packed-sorting schemes, at least in their basic form, tend to be value-sorting and not rank-sorting.

In a sequential setting, there is actually no real difference between value-sorting and rank-sorting, since the latter reduces to the former. Suppose that we are given an array of n integer keys, where n is a power of 2, to be rank-sorted. We replace the integer key x of the ith record by the value $nx + (i - 1)$, for $i = 1, \ldots, n$, and value-sort the resulting modified keys. Since the lower-order $\log n$ bits of each modified key contain the index of the array location storing the corresponding original key, we can rank-sort the original keys in $O(n)$ further time. The only snag in this argument is that while the original keys fit in one word, each modified key may occupy two words, with the most significant word holding a value in the range $0 \mathinner{\ldotp\ldotp} n-1$. In the sequential setting this is never an issue: We can sort the modified keys in two passes, value-sorting the lower-order words in the first pass, and then sorting the higher-order words by means of (stable) bucket sorting in $O(n)$ time. In the parallel setting we need to be more careful, as the resources needed for stable bucket sorting by the best known algorithms are not always negligible. However, we will appeal to the principle above only when the resources needed by the particular sorting routine employed are not affected (except to within a constant factor) by the addition of $\log n$ lower-order bits to each key, so that the two passes can be collapsed into one value-sorting pass; this will be the case when the original keys have $\Omega(\log n)$ bits each.

As a purely technical point, we assume a machine architecture that always allows us to address enough working memory for our algorithms, even when w is barely larger than $\log n$ (this is an issue only for $w = \log n + O(1)$, in which case bucket sorting works in linear time and space). Since we will always require that $w \geq \log n$, this allows us to assume without loss of generality that n and w are larger than arbitrary fixed constants. Furthermore, standard algorithms for multiple-precision arithmetic allow us to assume constant-time operations on words of $O(w)$ bits, rather than exactly w bits; put differently, the word length w is significant only up to a constant factor.

2 Sorting in $O(n \log \log n)$ time

Our goal in this section is to prove the following theorem.

Theorem 2.1 *For all given integers $n \geq 4$ and $w \geq \log n$, n integers in the range $0 \mathinner{..} 2^w - 1$ can be sorted in $O(n \log \log n)$ time on a unit-cost RAM with a word length of w bits and the restricted instruction set.*

For all positive integers n and b with $b \leq w$, denote by $T(n, b)$ the worst-case time needed to sort n integers of b bits each, assuming b and w to be known. A sequential version of a parallel algorithm due to Albers and Hagerup [2] shows that $T(n, b) = O(n)$ for all $n \geq 4$ and $b \leq \lceil w/(\log n \log \log n) \rceil$, i.e., provided that $\Omega(\log n \log \log n)$ keys can be packed into one word, sorting can be accomplished in linear time. This follows directly from Corollary 1 of [2]. (The corollary requires a quantity $\lfloor \log \log m \rfloor$ to be known, but it is easy to see that it suffices, in our case, to know the word length w.) We sketch the algorithm to illustrate its simplicity. It stores keys in the so-called *word representation*, i.e., k to a word, where $k = \Theta(\log n \log \log n)$, and its central piece is a subroutine to merge two sorted sequences, each consisting of k keys and given in the word representation, in $O(\log k)$ time. Essentially using calls of this subroutine instead of single comparisons, the algorithm proceeds as in standard merge sort to create longer and longer sorted runs. Since it can handle k keys at a cost of $O(\log k)$, it saves a factor of $\Theta(k/\log k)$ relative to standard merge sort, so that the total time needed comes to $O(n \log n \log k/k) = O(n)$. As described, the algorithm only value-sorts its input; recall from the introduction, however, that we can easily derive a rank-sorting algorithm with the same resource bounds.

Our second ingredient is the range reduction of Kirkpatrick and Reisch [26, Corollary 4.2], embodied in the recurrence relation

$$T(n, b) \leq T(n, \lceil b/2 \rceil) + O(n),$$

i.e., in $O(n)$ time we can reduce by about half the number of bits in the integers to be sorted; again, code realizing the reduction fits on one page.

Let us now prove Theorem 2.1. In order to sort n given keys, we first apply the range reduction of Kirkpatrick and Reisch $2\lceil \log\log n \rceil$ times, at a total cost of $O(n \log\log n)$. This leaves us with the problem of sorting n integers of at most $\lceil w/(\log n)^2 \rceil$ bits each, which can be done in $O(n)$ time using the algorithm of Albers and Hagerup.

3 Sorting in linear expected time

In this section we describe the *signature sort* algorithm and show that it works in linear expected time if w is sufficiently large. Signature sort is obtained by combining the packed-sorting algorithm of [2] with a new, randomized range-reduction scheme. We now admit multiplication as a unit-cost operation.

We first provide an informal sketch of the main ideas behind signature sort. Just as the algorithm of Kirkpatrick and Reisch reduces the length of the keys to be sorted by half in linear time, signature sort reduces their length by a factor of q in linear time, where q is chosen such that integers of $\Theta(q \log n)$ bits each can be sorted in linear time. In order to sort n b-bit keys, we split each key into q fields of b/q bits each and represent each value occurring in one or more fields by a unique *signature* of $\Theta(\log n)$ bits, obtained by applying a universal hash function to the value. The signatures of all fields in a key can be computed together in constant time, and their concatenation is an integer of $\Theta(q \log n)$ bits. After sorting the concatenated signatures of the input keys in linear time, we construct their path-compressed trie, with signatures considered as characters, also in $O(n)$ time. The trie is a tree with fewer than $2n$ edges. Each leaf corresponds to an input key, and each edge is associated with a *distinguishing* signature in a natural way. All that remains is to sort the "sibling" edges below each node in the tree by the original (b/q)-bit field values corresponding to their distinguishing signatures, since after this operation the sorted sequence of the n input keys can be read off the tree in a left-to-right scan. It again suffices to reduce the number of bits in the numbers to be sorted by a factor of $\Theta(\log n \log\log n)$. Since we can choose $q = \Theta(w/((\log n)^2 \log\log n))$, a constant depth of recursion and linear overall time is obtained if $w \geq (\log n)^{2+\epsilon}$, for fixed $\epsilon > 0$. This ends the informal sketch.

Besides the usual interpretation of the contents of words as integers, we will interpret words as representing sequences of integers or truth values (booleans). Which interpretation is intended for a given word will be expressed implicitly through the operations applied to the word. Our interpretation is parameterized by two integers $M, f \geq 2$. These will mostly be implicit; when wanting to make them explicit, we speak of the *(M, f)-representation*. When words intended to represent objects according to the (M, f)-representation are given as input to a routine, we will always assume that the integers M and f are made available as well.

The (M, f)-representation partitions the rightmost Mf bits of a word into M *fields* of f bits each, while ignoring any other bits present in the word. The fields are numbered $1, \ldots, M$ from right to left, and the leftmost bit of each field, called its *test bit*, is required to be zero. Suppose that field i of a word X contains the integer x_i, for $i = 1, \ldots, M$ (according to the usual binary representation). Then one interpretation of X is as the integer sequence $(x_1, \ldots, x_M)$. The interpretation of X as a boolean sequence additionally requires that $x_i \in \{0, 1\}$, for $i = 1, \ldots, M$, and interprets X as the sequence $(\tau(x_1), \ldots, \tau(x_M))$, where $\tau(1) = \mathit{true}$ and $\tau(0) = \mathit{false}$.

Building on and extending ideas of [31] and [18], we now develop an arsenal of basic operations, many of which operate on sequences of integers or booleans on a component-by-component basis. The built-in bitwise boolean operations will be denoted by AND and OR, and the shift operator is rendered as $\uparrow$ or $\downarrow$: When x and i are integers, $x \uparrow i$ denotes

$\lfloor x \cdot 2^i \rfloor \bmod 2^w$, and $x \downarrow i = x \uparrow (-i)$. In the following, assume the (M, f)-representation used throughout, for integers $M, f \geq 2$. As is common, we do not always distinguish between a variable and its value; e.g., we may write $X = (x_1, \ldots, x_M)$, where X is a variable and $(x_1, \ldots, x_M)$ is the sequence that it represents.

First, the constant $\sum_{i=0}^{M-1} 2^{if}$, which represents the sequence $1_{M,f} = (1, \ldots, 1)$, can be computed in $O(\log M)$ time by noting that $1_{2m,f} = 1_{m,f} \cdot (1 + 2^{mf})$, for all integers $m \geq 2$. As will be seen later, much of the utility of the constant $1_{M,f}$ comes from the fact that multiplication with $1_{M,f}$ carries out a prefix summation. Componentwise logical conjunction and disjunction, denoted $\wedge$ and $\vee$, are easy, since they may be implemented directly through AND and OR. Componentwise logical negation, denoted $\neg$, is just subtraction from $1_{M,f}$. As a slightly less trivial operation, consider $[X \geq Y]$, where $X = (x_1, \ldots, x_M)$ and $Y = (y_1, \ldots, y_M)$ are integer sequences, which returns the boolean sequence $(b_1, \ldots, b_M)$ with $b_i = \mathit{true}$ if and only if $x_i \geq y_i$, for $i = 1, \ldots, M$. $[X \geq Y]$ can be computed by subtracting Y from X after first setting all test bits in X to 1. The test bits prevent carries between fields, and the test bit in field i will "survive" exactly if $x_i \geq y_i$, so that all that remains is to shift the test bits to the rightmost position of the fields and to mask away all other bits. Thus $[X \geq Y]$ can be obtained as $((X + (1_{M,f} \uparrow (f-1)) - Y) \downarrow (f-1))$ AND $1_{M,f}$. Because the full range of componentwise boolean operators is available, it is an easy matter to implement the remaining componentwise relational operators $\leq$, $>$, $<$, $=$ and $\neq$. E.g., $[X = Y] = [X \geq Y] \wedge [Y \geq X]$. Another useful operator is the *extract* operator $|$. When $X = (x_1, \ldots, x_M)$ is an integer sequence and $B = (b_1, \ldots, b_M)$ is a boolean sequence, $X \mid B$ denotes the integer sequence $(y_1, \ldots, y_M)$ such that for $i = 1, \ldots, M$,

$$y_i = \begin{cases} x_i, & \text{if } b_i = \mathit{true}, \\ 0, & \text{if } b_i = \mathit{false}. \end{cases}$$

$X \mid B$ can be obtained simply as X AND $(B \cdot (2^f - 1))$.

Lemma 3.1 *Suppose that we are given two integers $M \geq 2$ and $f \geq \log M + 2$, a word X representing a sequence of integers according to the (M, f)-representation, and the constant $1_{M,f}$. Then, in constant time and using a word length of Mf bits, we can compute the index of the leftmost nonzero field in X (zero if there is no such field).*

PROOF Setting $A := [X > 0] \cdot 1_{M,f}$ computes for each field the number of nonzero fields to its right, including itself; the condition $f \ge \log M + 2$ ensures that the fields are wide enough to hold the counts. In particular, $m := (A \downarrow ((M-1)f))$ AND $(2^f - 1)$ is the total number of nonzero fields in X. Assume that $m > 0$. Then $B := [A = m \cdot 1_{M,f}] \wedge [X > 0]$ contains 1 in the field of interest and zeros in all other fields. Taking $C := (1_{M,f})^2 = (1, 2, \ldots, M)$ and forming $D := C \mid B$ replaces the 1 in the field of interest by the index of that field. The latter quantity, which is the desired answer, can finally be obtained as $((D \cdot 1_{M,f}) \downarrow ((M-1)f))$ AND $(2^f - 1)$. If $m = 0$, the same computation yields zero. □

We now return to the sorting problem and first give a high-level description of the new range reduction that ignores details such as rounding. Take $q = w/((\log n)^2 \log\log n)$ and assume that $q \ge 2$.

In order to sort n keys of b bits each, we begin by conceptually partitioning each key into q k-bit fields, where $k = b/q$. Assume that we are given a function $h : \{0, \ldots, 2^k - 1\} \to \{0, \ldots, 2^l - 1\}$, where $l = \Theta(\log n)$, that operates injectively on the set of all fields occurring in the input keys. We will actually consider the images under h as strings of $f = l + 1$ bits, with the leftmost bit always equal to zero. For each field x, we call $h(x)$ the *signature* of x; furthermore, if a key X consists of fields $x_1, \ldots, x_q$, we define the *concatenated signature* of X as the integer obtained by concatenating (the f-bit strings representing) the signatures $h(x_1), \ldots, h(x_q)$.

We now sort the n input keys by their concatenated signatures. By the choice of q, this can be done in linear time. Unless h happens to be monotonic, this arranges the keys in an order different from the one required by the original sorting problem, but one that nonetheless turns out to be useful.

Let $Y_1, \ldots, Y_n$ be the concatenated signatures in the order in which they appear after the sorting (i.e., $Y_1 \le Y_2 \le \cdots \le Y_n$) and take $\mathcal{Y} = \{Y_1, \ldots, Y_n\}$, formed as a multiset. Viewing the elements of $\mathcal{Y}$ as character strings of length q over the alphabet $\Sigma = \{0, \ldots, 2^f - 1\}$, we now aim to construct a *path-compressed trie* T_{D} for $\mathcal{Y}$. (For a more detailed discussion of the material that follows, consult [21].) If $Y_1, \ldots, Y_n$ are distinct, T_{D} is a tree with a leaf node for each element of $\mathcal{Y}$ and an internal node for each string over Σ that is the longest common prefix of two strings in $\mathcal{Y}$, and the parent of each nonroot node s in T_{D} is the longest proper prefix of s that occurs as a node in T_{D}. If $Y_1, \ldots, Y_n$ are not all distinct, we modify the definition of T_{D} slightly by considering identical strings as differing in a fictitious $(q+1)$st character; this ensures that each input key indeed corresponds to a separate leaf. We will assume that each internal node in T_{D} is marked with the length of the relevant common prefix, and that each leaf in T_{D} is marked with the corresponding input key (of which the leaf is the concatenated signature); after an easy computation, we can assume that each internal node s in T_{D} is marked with one of the input keys occurring in the subtree rooted at s.

In order to construct T_{D}, we begin by computing the length r_i of the longest common prefix of Y_i and Y_{i+1}, for $i = 1, \ldots, n-1$; by Lemma 3.1, applied to words of the form $[Y_i \neq Y_{i+1}]$, this can be done in a total time of $O(n)$. We then construct what is known as the *Cartesian tree* of the sequence $(r_1, 1), \ldots, (r_{n-1}, n-1)$, where pairs are compared lexicographically. The Cartesian tree of a sequence $a_1, \ldots, a_m$ of distinct elements drawn from a totally ordered universe is the (possibly empty) tree T defined inductively as follows: (1) If $m = 0$, T is the empty tree; (2) If $m \geq 1$, the root of T is $a_{i_0} = \min\{a_1, \ldots, a_m\}$, and the two subtrees of the root are the Cartesian trees of the sequences $a_1, \ldots, a_{i_0-1}$ and $a_{i_0+1}, \ldots, a_m$. Gabow et al. [20] showed that Cartesian trees of given sequences can be constructed in linear time, so that we obtain the Cartesian tree T_{C} of $(r_1, 1), \ldots, (r_{n-1}, n-1)$ in $O(n)$ time. As observed in [21], if we add two children below each leaf in T_{C}, we obtain a tree that is isomorphic to the compressed trie T_{D}, except for the fact that T_{C} is always binary, while T_{D} is not: Each node in T_{D} with $d \geq 3$ children is represented in T_{C} by a path of $d-1$ internal nodes connected to descendant via exactly d edges. Since this difference is computationally trivial (indeed, going from T_{C} to T_{D} can be viewed as merely a matter of interpreting the representation in a different way), it is easy to see that we can obtain T_{D} in linear time.

The crucial observation at this point is that we can sort the input keys, attached to the leaves of T_{D}, by sorting the children of each internal node in T_{D} by the original fields corresponding to the distinguishing signatures in which they differ. Since the information available locally in the tree suffices to construct a list of the fields concerned in linear time for each internal node, we are now faced with the problem of sorting a total of $n+g-1 < 2n$ fields within disjoint groups, where g is the number of internal nodes in T_{D}. Using an idea proposed by Kirkpatrick and Reisch [26], we can reduce the total number of fields to be sorted below n by removing the maximum field within each of the disjoint groups and adding it back after the sorting; this, of course, needs only $O(n)$ time. The original problem of sorting n keys of b bits each is thus reduced to that of sorting fewer than n keys of b/q bits each within disjoint groups. All that remains is a left-to-right traversal of T_{D}, during which the input keys are output as they are encountered. The idea of first constructing an unordered compressed trie and then sorting at each of its internal nodes was also used in [30] and [5].

We still need to describe how to obtain and evaluate the function $h : \{0, \ldots, 2^k - 1\} \to \{0, \ldots, 2^l - 1\}$. Recall that what we require of h is that it must operate injectively on a set S of nq fields. Provided that l is sufficiently large, we can ensure this, with high probability, by choosing h at random from a suitable class of hash functions. In fact, most reasonable classes of hash functions have this property (the class should be what is known as *universal*), but we are severely restricted in our choice of hash functions by the facts that, first, our instruction repertoire does not include division and, second, we can spend only constant time computing the signatures of all fields in a word. A class of hash functions that fits the bill is the class $\mathcal{H} = \{h_a \mid 0 < a < 2^k, \text{ and } a \text{ is odd}\}$, where h_a is defined by

$$h_a(x) = (ax \bmod 2^k) \operatorname{div} 2^{k-l},$$

for $x = 0, \ldots, 2^k - 1$. It can be seen that h_a simply picks out a segment of l consecutive bits from the product ax. In order to compute the signatures of all fields in a word in constant time, we treat the fields in even-numbered positions and those in odd-numbered positions separately. To obtain the signatures of all even-numbered fields, we first clear the fields in odd-numbered positions (i.e., they are set to zero) by means of a suitable mask, which creates "buffer zones" between the fields of interest. The whole resulting word is then multiplied by a, the buffer zones preventing overflow from one field from interfering with the multiplication in another field, and the final application of a suitable mask clears all bits outside of the signatures. At this point the signatures of the even-numbered positions are easily combined with those of the odd-numbered positions. Note that the integer represented by the word computed so far, although closely related to the concatenated signature of the original key, is essentially as large as the original key—each signature, though only f bits long, still occupies a k-bit field, with zeros in unused bit positions. The signatures can be packed tightly in adjacent f-bit fields by means of a simple extension of the algorithm of [18, Lemma 3] shown in Fig. 1. This takes constant time, since all that is involved is a multiplication, a shift and a masking operation. The compaction works correctly only if the number of fields per word is bounded by the ratio of the width of a field to the width of a signature, so that a problem arises for $b < q^2 f$. In this case, however, it suffices to divide a word into $b/(qf)$ fields, rather than q fields; the resulting fields of qf bits each can be sorted directly in linear time, so that no recursive invocation is needed.

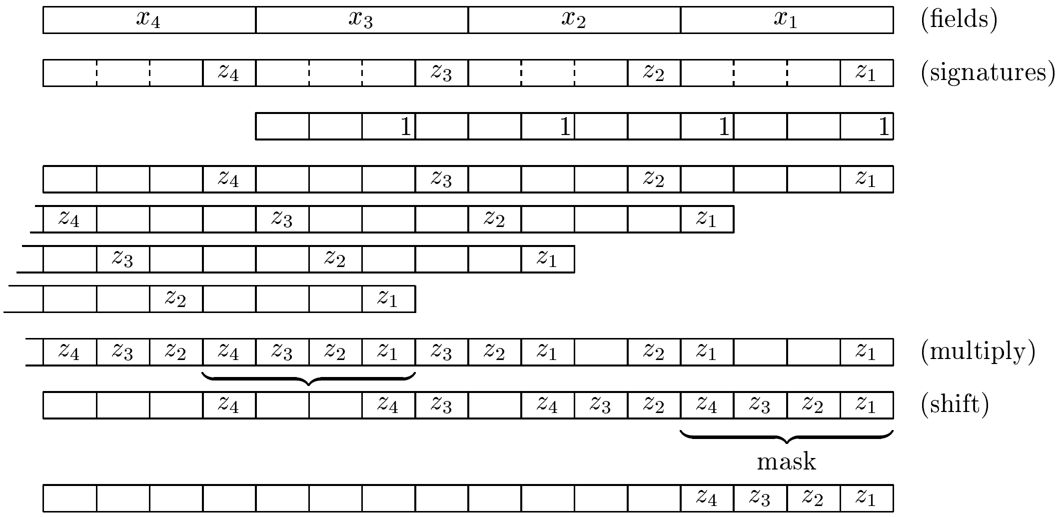

Figure 1: Computing concatenated signatures. For $i = 1, \ldots, 4$, $z_i = h(x_i)$.

The class $\mathcal{H}$ was analyzed by Dietzfelbinger et al. [15], who establish (Lemma 2.3) that if h is chosen randomly from $\mathcal{H}$ (which amounts to choosing the multiplier a at random), then h is injective on S with probability at least $1 - |S|^2/2^l$. We can assume without loss of generality that $q \le n$ and hence $|S| \le n^2$, since even for $q = \Theta(\log n \log\log n)$ the problems generated by the reduction can be solved in linear time. Thus we can make the probability that h is not injective on S smaller than $1/n^2$ by choosing $l = \Theta(\log n)$ appropriately.

This completes the description of the randomized reduction. The original problem is reduced in $O(n)$ time to that of sorting fewer than n keys that are a factor of q shorter than the original input keys, or that can be sorted in linear time, within disjoint groups. For $n \geq 1$, $1 \leq b \leq w$ and $0 < p \leq 1$, denote by $T(n, b, p)$ the time needed to sort at most n integers of b bits each within disjoint groups with probability at least p, assuming b and w to be known. The reduction can be summarized in the recurrence relation

$$T(n, b, p) \leq T(n, b/q, p + 1/n^2) + O(n).$$

As in Section 2, we apply the reduction repeatedly until the remaining sorting problem can be solved directly using the algorithm of Albers and Hagerup, i.e., until the length of the numbers involved has dropped by a factor of $\Theta(\log n \log\log n)$; it is easy to see that this happens after $O(1 + \log\log n/\log q)$ reduction steps. Disregarding a few technicalities that were ignored above and will be dealt with below, this proves the following main result.

Theorem 3.1 *For all given integers $n \geq 4$ and $w \geq 2(\log n)^2 \log\log n$, a unit-cost RAM with a word length of w bits and the full instruction set can sort n integers in the range $0\,..\,2^w - 1$ in*

$$O(n + n \log\log n/\log q)$$

time, where $q = w/((\log n)^2 \log\log n)$, with probability at least $1 - 1/n$.

Proof We use the algorithm described above, but still need to give some more details of the choice of parameters and the computation of concatenated signatures.

Begin by choosing $l = \Theta(\log n)$ sufficiently large for the analysis (e.g., since we can assume that $|S| \leq n^2$, $l > 6 \log n$ will do) such that $f = l + 1$ is a power of 2. The remaining part of the detailed implementation depends on the size of w relative to n.

Assume first that $w \leq \lfloor \log n \rfloor^4$. Then we can easily compute with w, i.e., derive from w the auxiliary parameters needed by the algorithm. Begin by determining $\lceil \log w \rceil$ and replace w by $2^{\lceil \log w \rceil}$, i.e., pretend that w is a power of 2; this at most doubles the necessary word length. We will choose several other parameters as powers of 2; we always assume that their logarithms are computed as well, so that divisions by these parameters can be realized in constant time as right shifts. Briefly letting $q' = w/((\log n)^2 \log\log n)$, choose q as a power of 2 with $q \geq 2q' \geq 4$, but $q = O(q')$. Now integers of qf bits can be sorted in linear time, as required.

Consider the problem of sorting b-bit integers, where $b \leq w$ is a power of 2, and take $m = \min\{q, b/(qf)\}$ (the number of fields) and $k = b/m$ (the field width). We can assume that $m \geq 4$, since otherwise the sorting can be carried out directly in linear time. Given a random multiplier a, the computation of the concatenated signature of a b-bit integer stored in a word X can take the following form: First appropriate masks $B_1 := 1_{m/2,2k} \cdot (2^k - 1)$ and $B_2 := 2^b - 1 - B_1$ are computed, then $C := \sum_{i=1}^{2}((a \cdot (X \text{ AND } B_i)) \text{ AND } B_i)$ essentially carries out the multiplication with a, and $D := (C \downarrow (k - l)) \text{ AND } (1_{m,k} \cdot (2^l - 1))$ leaves only the signatures of the fields. In order to pack the signatures tightly according to Fig. 1, we compute $E := D \cdot 1_{m,k-f}$ and $F := E \downarrow ((m-1)(k-f))$ and obtain the concatenated signature as $F \text{ AND } (2^{mf} - 1)$. Note that the relation $mf \leq k$ ensures that fields do not "collide" during the compaction.

Since all but the last reduction step reduces the number of bits in the integers to be sorted by a factor of q, the number of reduction steps will be $O(1 + \log\log n/\log q)$, for a total time of $O(n + n\log\log n/\log q)$. Each of the constants of the form $1_{\alpha,\beta}$ needed by the algorithm can be computed in $O(\log\log n)$ time, and the total number of such constants needed over all reduction steps is $O(\log\log n)$, so that their computation is not a bottleneck.

Suppose now instead that $w > \lfloor \log n \rfloor^4$. The main difference to the case of small values of w is that we cannot easily compute with w and that only a single reduction step is needed. We begin by computing integers $k \geq 1$ and $m \geq 4$ such that m is even, $k = \Theta(w/(\log n \log\log n))$, $mk \geq w$, $mk = O(w)$, and $mf \leq k$. For sufficiently large values of n, such integers exist and can be computed in $O(\log n)$ time; e.g., k can be obtained from w via a right shift by approximately $\log(\log n \log\log n)$ bits. Taking $b = mk$, we can compute concatenated signatures exactly as in the case $w \leq \lfloor \log n \rfloor^4$. The correctness is again guaranteed by the relation $mf \leq k$, and since the fields can be sorted in linear time, only a single reduction step is needed. The necessary constants of the form $1_{\alpha,\beta}$ can be computed in $O(\log \log n)$ time, and the complete sorting finishes in $O(n)$ time. $\square$

For large values of q, the failure probability of $1/n$ indicated in Theorem 3.1 is not the best possible. We shall not elaborate on this, but mention that a smaller probability is obtained simply by choosing a larger value of l. The same remark applies to Theorem 7.1.

Corollary 3.1 *If $w \geq (\log n)^{2+\epsilon}$ for some fixed $\epsilon > 0$, we can sort in linear expected time.*

4 Sorting multiple-precision integers

Building on ideas of [26] and [30], an algorithm of Andersson and Nilsson [5] reduces the problem of sorting n multiple-precision integers occupying a total of N words to that of sorting n (single-precision) integers; the reduction needs $O(N + n)$ time. Combining this with Theorem 2.1 and Corollary 3.1, we obtain two algorithms for the general lexicographic sorting problem.

Corollary 4.1 *For all integers $n, N \geq 4$, n multiple-precision integers occupying a total of N machine words can be sorted in $O(N + n \log \log n)$ time or, provided that $w \geq (\log n)^{2+\epsilon}$ for some fixed $\epsilon > 0$, in $O(N + n)$ expected time.*

5 Space requirements

As is easy to discover from an inspection of the algorithms of [2] and [26], the deterministic algorithm of Section 2 works in $O(2^w)$ space. The only point that might need clarification concerns the recursion stack needed for successive range-reduction steps. The only information that needs to be pushed on the recursion stack is the input, a list of n numbers, since all other information, rather than being preserved across the recursive call, can be recomputed from the input after the call. Moreover, the number of bits needed to store the input numbers is reduced by a factor of essentially 2 from one reduction step to the next. By storing several numbers in each machine word, we can therefore arrange that the total space taken up by the recursion stack is $O(\sum_{i=0}^{\infty} n/2^i) = O(n)$. Thus the space requirements of the stack are dominated by the $O(2^w)$ space needed by the rest of the algorithm.

By breaking each input key into r pieces of at most $\lceil w/r \rceil$ bits each, for some $r \geq 2$, thereby in effect reducing the word length, and sorting the resulting multiple-precision integers as described in the previous section, we obtain a sorting algorithm that uses $O(nr + n \log\log n)$ time and $O(n + 2^{w/r})$ space. (The reduction of Corollary 4.1 itself works in $O(n + 2^{w/r})$ space.)

If we are willing to accept randomization and use the full instruction set, we can obtain a sorting algorithm that runs in $O(n \log\log n)$ expected time using $O(n)$ space by means of universal hashing. This follows easily from [11, Proposition 3] and the observation that each range-reduction step works in $O(n)$ time. In order to avoid the use of division, we employ the class of universal hash functions of [15]. A stronger result of this kind is implied by the considerations in Section 9.

The recursion stack of signature sort can also be represented in linear space, so that signature sort works in $O(n)$ space.

6 Deterministic parallel sorting

We begin this section by discussing two parallel packed-sorting algorithms. One is the original algorithm of Albers and Hagerup, whose performance bounds are restated for the reader's convenience. The second algorithm is new and based on a combination of the sorting network of Ajtai et al. [1] and the packed-sorting algorithm of Paul and Simon [31]. We subsequently describe a parallel version of the range reduction of Kirkpatrick and Reisch and combine it with the packed-sorting algorithms to obtain new algorithms for conservative sorting.

Lemma 6.1 *For all given integers $n \geq 4$ and $w \geq \log n$, n integers of $\lceil w/(\log n \log\log n) \rceil$ bits each can be value-sorted in $O((\log n)^2)$ time using $O(n)$ operations on an EREW PRAM with the restricted instruction set. On the CREW PRAM, the same result holds, except that the running time is $O(\log n \log\log n)$.*

PROOF The first part of the lemma is just Corollary 1 of [2]. It turns out that the only part of the algorithm of that corollary that needs more than $\Theta(\log n \log\log n)$ time are $\Theta(\log n)$ successive rounds of merging longer and longer sorted runs of input numbers. The second part of the lemma follows by observing that merging can be done in doubly-logarithmic time on the CREW PRAM [28]. □

The algorithms of Lemma 6.1 need more than logarithmic time because they are based on repeated merging. We now provide an alternative algorithm that value-sorts n keys in $O(\log n)$ time, but in return requires more keys to fit in one word and needs multiplication.

For all integers $M, f \geq 2$, we extend the (M, f)-representation to cover objects of one additional type, namely multisets of integers. If field i of a word X contains the integer x_i, for $i = 1, \ldots, M$, X may be interpreted as the multiset obtained from the multiset $\{x_1, \ldots, x_M\}$ by removing all occurrences of zero; in other words, a field with a value of zero is interpreted as being "empty". We sometimes restrict the multiset representation further by requiring all nonzero field values to be distinct; in this case we will call the object represented a (simple) set, rather than a multiset.

Lemma 6.2 *Suppose that we are given two integers $M \geq 2$ and $f \geq \log M + 2$, a word X representing a simple set U according to the (M, f)-representation, an integer r with $1 \leq r \leq |U|$, and the constants $1_{M,f}$, $1_{M,Mf}$ and $1_{M,(M-1)f}$. Then, in constant sequential time and using a word length of $M^2 f$ bits, we can find the element of U whose rank in U is r.*

PROOF Denote by x_i the integer contained in field i of X, for $i = 1, \ldots, M$. We will temporarily adopt the (M^2, f)-representation, i.e., operations like $|$ are to be interpreted accordingly below, and we begin by replacing X by X AND $(2^{Mf} - 1)$ in order to remove any spurious bits; note that the fundamental constant $1_{M^2,f}$ can be obtained as $1_{M,f} \cdot 1_{M,Mf}$. The basic idea, which goes back to Paul and Simon [31], is to create words A and B such that field number $(j-1)M + i$ of A contains x_i, while the corresponding field of B contains x_j, for $i = 1, \ldots, M$ and $j = 1, \ldots, M$, and then to carry out all pairwise comparisons between elements of $\{x_1, \ldots, x_M\}$ by evaluating $[A \geq B]$ (see Fig. 2). A is easily computed

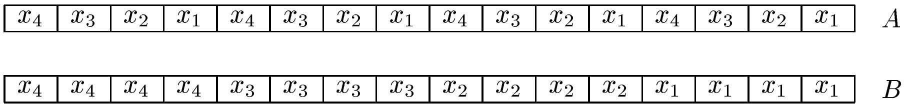

Figure 2: The words A and B.

as $X \cdot 1_{M,Mf}$, and B can be obtained as $((X \cdot 1_{M,(M-1)f}) \mid 1_{M,Mf}) \cdot 1_{M,f}$ (see Fig. 3).

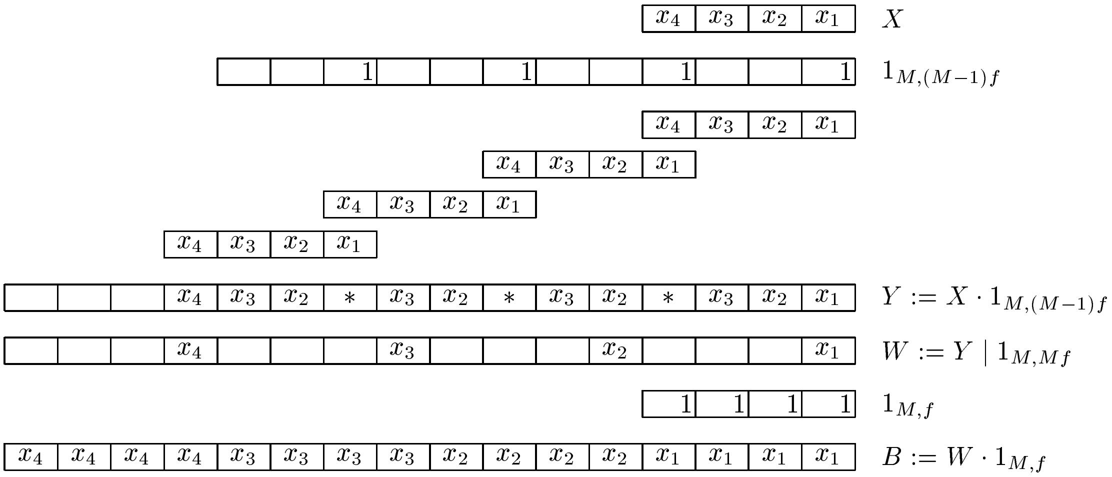

Figure 3: Stages in the computation of B. $*$ denotes a don't-care value.

Setting $C := (([A \geq B] \wedge [B > 0]) \cdot 1_{M,Mf}) \downarrow ((M-1)Mf)$ computes the rank of x_i in U and stores it in field i of the (M,f)-representation, for $i = 1, \ldots, M$, provided that $x_i \neq 0$ (see Fig. 4). Recall that if $x_i = 0$, then, by definition, $x_i \notin U$, and note how the test $B > 0$ prevents zero elements of $x_1, \ldots, x_M$ from interfering with the rank computation. As in the algorithm of Lemma 3.1, the condition $f \geq \log M + 2$ ensures that fields are wide enough to hold the ranks.

We now revert to the (M,f)-representation and remove all elements of U except the one of rank r by setting $D := X \mid [C = r \cdot 1_{M,f}]$. Finally the element of rank r is obtained as $((D \cdot 1_{M,f}) \downarrow ((M-1)f))$ AND $(2^f - 1)$. $\square$

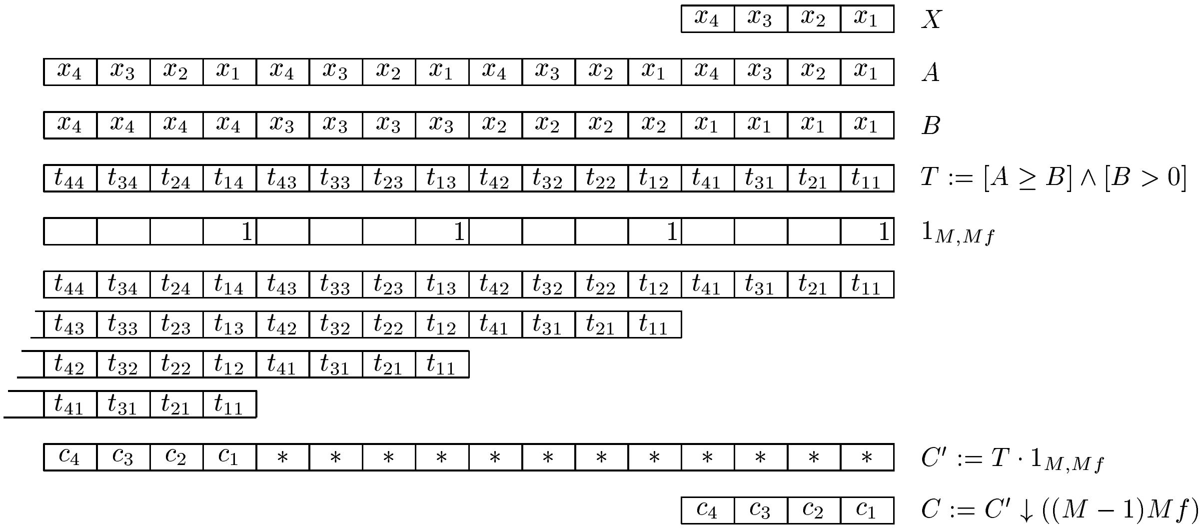

Figure 4: Computing the ranks of the x_i's. t_{ij} denotes the result of the comparison $x_i \overset{?}{\geq} x_j$, for $i = 1, \ldots, M$ and $j = 1, \ldots, M$, and c_i is the rank of x_i, for $i = 1, \ldots, M$. $*$ represents a don't-care value.

Given two multisets U and V of integers containing the same number k of elements, we denote by $U \Lleftarrow\!\!\!\!\wedge V$ and $U \vee\!\!\!\vee V$ the multisets consisting of the k smallest and the k largest elements of the $(2k)$-element multiset $U \cup V$, respectively. We will use the term "k-halver" to denote a device that inputs two multisets U and V of k integers each and outputs $U \Lleftarrow\!\!\!\!\wedge V$ and $U \vee\!\!\!\vee V$. The following lemma describes the implementation of a k-halver.

Lemma 6.3 *Suppose that we are given integers $M \geq 2$, $m = \lceil \log M \rceil + 2$, $f \geq m + 1$, and $k \geq 1$, two words X and Y representing multisets U and V of cardinality k each according to the (M, f)-representation, and the constants $1_{M,f}$, $1_{2M,2Mf}$, and $1_{2M,(2M-1)f}$. Suppose further that the $m + 1$ most significant bits of each field of X and Y are zero. Then, in constant sequential time and using a word length of $4M^2 f$ bits, we can compute words representing $U \Lleftarrow\!\!\!\!\wedge V$ and $U \vee\!\!\!\vee V$ according to the $(2M, f)$-representation.*

Proof We first combine X and Y by computing $W := (X \text{ AND } (2^{Mf} - 1)) + (Y \uparrow (Mf))$. From now on we employ the $(2M, f)$-representation. The idea is simply to split the multiset stored in W at its median, the latter being found with the algorithm of Lemma 6.2. Before we can appeal to Lemma 6.2, however, we have to convert the multiset stored in W to a simple set by imposing a total order among equal elements. We do this by shifting each element left by m bits and appending a unique marker to the right end of each element. By the assumption of free leading bit positions in each field, the representation remains valid, and the relative order of distinct elements is as before, which will ensure the correctness of the procedure. The unique end markers are obtained from the word $A = (1_{2M,f})^2 = (1, 2, \ldots, 2M)$, so that altogether we execute $W := (W \uparrow m) + (A \mid [W > 0])$. Now we can employ the algorithm of Lemma 6.2 to determine the element x of rank k. Subsequently we compute the two words $W \mid [W \leq x \cdot 1_{2M,f}]$ and $W \mid [W > x \cdot 1_{2M,f}]$ and return them after removing their end markers and shifting them right by m bits (while removing any spurious bits introduced by the right shift). $\square$

An important fact to note about the lemma above is that the output is "less compact" than the input, in that the number of fields per word has doubled, while the number of nonempty fields per word remains exactly the same. In order to counteract this drift, we will regularly compact words representing multisets in the sense described in the following lemma.

Lemma 6.4 *Given two integers $M \geq 2$ and $f \geq \log M + 2$ and a word X representing a multiset U according to the (M, f)-representation, a word representing U according to the $(\max\{|U|, 2\}, f)$-representation can be computed sequentially in $O(\log M)$ time using a word length of Mf bits.*

PROOF We adapt a classical algorithm developed in the context of routing on hypercubic networks. We first give a high-level description of the algorithm and then describe its detailed implementation.

The goal will be to pack the elements of U tightly without changing the relative order in which they occur in X. Hence for $i = 1, \ldots, M$, if field i contains an element that has r_i zero fields to its right, then this element should be moved right by r_i field widths—call r_i its *move distance*. The actual movement takes place in $\lceil \log M \rceil$ phases. In Phase t, for $t = 0, \ldots, \lceil \log M \rceil - 1$, some elements move right by 2^t field widths, while the other elements remain stationary. Whether or not an element should participate in the movement in Phase t can be read directly off the corresponding bit of its move distance. The nontrivial fact about the algorithm, which guarantees its correctness, is that fields never "collide" during the movement (see, e.g., [29, Section 3.4.3]).

The sequence $R = (r_1, \ldots, r_M)$ of move distances is computed by the instruction $R := [X = 0] \cdot 1_{M,f}$, and the movement in Phase t simply computes $A := (R \downarrow t)$ AND $1_{M,f}$ and replaces X by $(((X \mid A)$ AND $(2^{Mf} - 1)) \downarrow (2^t f)) + (X \mid \neg A)$, for $t = 0, \ldots, \lceil \log M \rceil - 1$. $\square$

For our purposes, a *comparator network* of *width* $m \in \mathbb{N}$ is a straight-line program consisting of a sequence of instructions of the form $\mathit{Compare}(i, j)$, where $1 \le i < j \le m$. The intended semantics is that a comparator network of width m operates on an array $Q[1 \mathinner{..} m]$ containing m (not necessarily distinct) elements drawn from an ordered universe, and that the execution of an instruction $\mathit{Compare}(i, j)$ simultaneously replaces $Q[i]$ and $Q[j]$ by $\min\{Q[i], Q[j]\}$ and $\max\{Q[i], Q[j]\}$, respectively. If executing a comparator network $\mathcal{P}$ according to this interpretation sorts Q, i.e., if $Q[1] \le Q[2] \le \cdots \le Q[m]$ after the execution of $\mathcal{P}$ irrespectively of the initial contents of Q, $\mathcal{P}$ is called a *sorting network*. A *leveled* network of *depth* d is a comparator network whose sequence of *Compare* instructions is partitioned into d contiguous subsequences, called *levels*, such that no integer occurs more than once as an argument to *Compare* within a single level. All *Compare* instructions within one level of a leveled sorting network can clearly be executed in parallel without affecting the sorting property of the network. For all integers $m \ge 2$, the AKS network [1] is a leveled sorting network of width m and depth $O(\log m)$.

Let m and k be positive integers and suppose that we re-interpret a sorting network $\mathcal{P}$ of width m as follows: Rather than single elements, the cells of Q now contain multisets of k elements each, and the execution of $\mathit{Compare}(i, j)$ simultaneously replaces $Q[i]$ and $Q[j]$ by $Q[i] \Cap Q[j]$ and $Q[i] \Cup Q[j]$, respectively. Suppose further that we add to the beginning of $\mathcal{P}$ instructions to partition km elements arbitrarily into m multisets of k elements each and to store these in $Q[1], \ldots, Q[m]$, and that we add to the end of $\mathcal{P}$ instructions to sort the multiset $Q[i]$ into nondecreasing order, for $i = 1, \ldots, m$, and to concatenate the resulting sorted sequences in the order corresponding to $Q[1], \ldots, Q[m]$. We will call the procedure obtained in this way the *k-halving version* of $\mathcal{P}$. It is known that the k-halving version of any sorting network of width m sorts any sequence of km elements correctly [27, Exercise 5.3.4.38]; we will later prove a more general statement (Lemma 7.6).

Lemma 6.5 *For all given integers $n \geq 2$ and $w \geq \log n$ and all fixed $\epsilon > 0$, n integers of $b = \lceil w/(\log n)^{2+\epsilon} \rceil$ bits each can be value-sorted in $O(\log n)$ time using $O(n)$ operations on a unit-cost EREW PRAM with a word length of w bits and the full instruction set.*

PROOF Let k be the smallest power of 2 larger than $\log n$ and assume without loss of generality that k divides n and that $n, b \geq 4$. We will use the k-halving version of the AKS network $\mathcal{P}$ of width $m = n/k$, with each *Compare* instruction being executed by the k-halver of Lemma 6.3. Since the k-halver works in constant time and the depth of $\mathcal{P}$ is $O(\log m) = O(\log n)$, the sorting runs in $O(\log n)$ time. Furthermore, since the number of *Compare* instructions in a leveled comparator network cannot exceed the product of its width and depth, the total number of k-halving steps and, hence, the total number of operations executed is $O(m \log m) = O(n)$. What remains is to check a number of details.

Given n integers of b bits each and any integer $f > b$, it is a trivial matter, spending $O(k) = O(\log n)$ time and $O(n)$ operations, to partition the input numbers into m multisets of k elements each and to store each of these in a word according to the (k, f)-representation. One small complication derives from the fact that the value zero, stored in a field, is reserved to denote an "empty" field. We can deal with this by adding 1 to each key for the duration of the sorting, which may increase b by 1. This realizes the "preprocessing" of the k-halving version of $\mathcal{P}$. Similarly, the "postprocessing" can be realized by first converting each multiset, stored in the (k, f)-representation, to the corresponding sequence of k integers, stored in k words, and then sorting this sequence with the first algorithm of Lemma 6.1. The sorting needs $O((\log k)^2) = O(\log n)$ time and a total of $O(n)$ operations. Recall, however, that since the k-halvers of the k-halving version of $\mathcal{P}$ are implemented via Lemma 6.3, each level of the network blows up the representation by a factor of 2, i.e., takes us from the (M, f)-representation to the $(2M, f)$-representation, for some $M \geq k$. We need to limit the maximum value $M_{\max}$ of M that arises during the sorting, which we do by compacting the words produced by regularly spaced levels of the network. More precisely, for an integer $d \geq 1$, we compact the words at hand whenever the total number of levels executed so far is divisible by d, as well as after the final level. We choose d such that $d \leq \lceil (\epsilon/4) \log\log n \rceil$, but $d = \Omega(\log\log n)$. The first condition ensures that $M_{\max}$ is bounded by $k \cdot 2^{\lceil (\epsilon/4)\log\log n \rceil} \leq 2k(\log n)^{\epsilon/4} = O((\log n)^{1+\epsilon/4})$. In particular, since $M_{\max}$ is polylogarithmic in n, each compaction according to Lemma 6.4 takes $O(\log\log n)$ time, together with which the second condition imposed on d implies that the total time spent on compaction is within a constant factor of the depth of the network, i.e., negligible.

The word length needed is $4M_{\max}^2 f$ bits (Lemma 6.3 is the bottleneck). By the discussion above, this is $O(f(\log n)^{2+\epsilon/2})$ bits. According to Lemma 6.3, f must be chosen so large that each field, in addition to the b bits of the key stored there, has at least $\lceil \log M_{\max} \rceil + 3$ leading zero bits. Since $M_{\max}$ is polylogarithmic in n, we can easily satisfy this requirement while ensuring that $f = O(b(\log n)^{\epsilon/2})$; the necessary word length therefore is $O(b(\log n)^{2+\epsilon}) = O(w)$ bits, as promised. Note also that it is trivial to compute the constants of the form $1_{\alpha,\beta}$ required by Lemma 6.3 in $O(\log n)$ sequential time for all relevant values of M.

We now come to the construction of the AKS network itself. We follow the description given in [32], except that we replace the family of expanding graphs used there by the family of graphs given by Gabber and Galil [19] in order to simplify the task of constructing the network, at the expense of increasing the constant factor hidden in the O-notation. The key building blocks of the AKS construction are networks of constant depth and size $O(s)$ for "approximately classifying" two arrays containing $s \leq m/2$ elements each, for various values of s. If $s = l^2$ for an integer l, such a network can be constructed in constant time with $O(s)$ operations using the graphs of [19], provided that we can add two numbers modulo l in constant time. It turns out that the resulting network can also be used to approximately classify arrays of size $s' < s$, provided that $s' = \Omega(s)$. Accordingly, it suffices to consider only values of s of the form 4^j, for an integer j, in which case the remainder modulo l can be formed via a suitable masking operation.

Constructing the AKS network from these building blocks is also easy. The AKS network can be viewed as moving the elements of an array of size m among the nodes of a complete binary tree of height $\Theta(\log m)$. Say that a single time step is the depth required to approximately classify two arrays. A time step thus represents a constant number of levels in the sorting network. The AKS network runs for $T = \Theta(\log m)$ time steps and will be constructed in T stages, where in the tth stage the comparisons performed in the tth time step will be output in constant time by m EREW PRAM processors, for $t = 0, \ldots, T-1$. In order to do this it is necessary for each processor to know two quantities $a(t)$ and $b(t)$, which can be computed from $a(t-1)$ and $b(t-1)$ in constant time by one processor using the restricted instruction set, by keeping track of the value of t modulo 3 and modulo 4 ($a(0) = b(0) = 0$). Given the quantities $a(t)$ and $b(t)$ and the mapping of array locations to tree nodes in time step $t-1$, we can compute the mapping of array locations to tree nodes in time step t as well as various auxiliary quantities, which suffice to determine the required comparisons (the mapping of array locations to tree nodes in time step 0 is a trivial one). $\square$

The range reduction of Kirkpatrick and Reisch does not lend itself to easy direct parallelization. Bhatt et al. [9] discovered a way around this based on reducing the integer-sorting problem to another problem known as *ordered chaining* and applying parallel versions of the techniques of Kirkpatrick and Reisch to the latter problem. Here we exploit the connections between integer sorting and ordered chaining further: We first transform the initial integer-sorting problem to a problem of ordered chaining, then use the techniques of Kirkpatrick and Reisch and Bhatt et al. to transform the initial ordered-chaining problem to a collection of smaller ordered-chaining problems. At the end the collection of ordered-chaining problems is transformed back to a single integer-sorting problem.

The definition of ordered chaining below is equivalent to the one used by Bhatt et al. [9], but we prefer to phrase it in the style of [16].

Definition *For all integers $N \geq 1$, the ordered-chaining problem of size N is, given N processors numbered $0, \ldots, N-1$, some of which are (permanently) inactive (i.e., they do not take part in the computation), to compute for each active processor the smallest integer larger than its own number, if any, that is the number of an active processor.*

Informally, the ordered-chaining problem thus is to hook the active processors together in a linked list, sorted by processor number. It is important to note that the inactive processors are not available to participate in the computation. For N a power of 2 whose logarithm is known, Bhatt et al. describe a reduction of an ordered-chaining problem of size N to $2^{\lceil(\log N)/2\rceil}+1$ ordered-chaining problems, each of size at most $2^{\lceil(\log N)/2\rceil}$; the reduction takes constant time on a CRCW PRAM.

The transformation from integer sorting to ordered chaining exploited by Bhatt et al. is very simple: Assume that the initial task is to sort n integers $x_1,\ldots,x_n$ of w bits each, where n is a power of 2. We then associate a processor with x_i, for $i=1,\ldots,n$, and let it simulate an active processor with processor number $nx_i+(i-1)$, the inactive processors being entirely fictitious; this defines an instance of the ordered-chaining problem of size $2^{w+\log n}$. Applying the reduction above r times, for some $r\ge 1$, takes $O(r)$ time and yields a collection of ordered-chaining problems, each of size at most $2^{\lceil(w+\log n)/2^r\rceil}$. At this point each active processor generates a pair (i,j), where i is an identification of the subproblem to which the processor belongs, and j is its processor number within that subproblem. It is easy to see that sorting the pairs lexicographically solves all ordered-chaining problems in the collection. Undoing the reductions, we then obtain a solution to the original ordered-chaining problem in the form of a linked list. A solution to the original integer-sorting problem, finally, can be obtained in $O(\log n/\log\log n)$ time using $O(n)$ operations by applying the list-ranking algorithm of [9, Theorem 2] to this list.

Since the identification of an ordered-chaining problem can be taken as the real processor number of a processor simulating an active processor in the group, an integer in the range $1\,..\,n$, the argument above reduces the problem of sorting n integers of w bits each to that of sorting n integers of at most $\lceil\log n\rceil+\lceil(w+\log n)/2^r\rceil$ bits each; the reduction uses $O(r+\log n/\log\log n)$ time and $O(nr)$ operations on a CRCW PRAM.

Theorem 6.1 *For all given integers $n\ge 4$ and $w\ge\log n$, n integers in the range $0\,..\,2^w-1$ can be sorted in $O(\log n\log\log n)$ time using $O(n\log\log n)$ operations on a unit-cost CRCW PRAM with a word length of w bits and the restricted instruction set. With the full instruction set, the same result holds, except that the running time is $O(\log n)$.*

Proof We use the reduction given above with $r = 3\lceil \log\log n \rceil$, which takes $O(\log n / \log\log n)$ time and uses $O(n \log\log n)$ operations. The resulting problem of sorting n integers of at most $\lceil \log n \rceil + \lceil (w + \log n)/(\log n)^3 \rceil$ bits each is solved using the algorithm of [9] if $w \le \lfloor \log n \rfloor^4$, and using the packed-sorting algorithm either of the second part of Lemma 6.1 or of Lemma 6.5 otherwise. The packed-sorting subroutine used if $w > \lfloor \log n \rfloor^4$, though only value-sorting, is easily turned into a rank-sorting algorithm, as required here, since it is applied to keys of $\Omega(\log n)$ bits each. $\square$

7 Randomized parallel sorting

In this section we first describe a randomized packed-sorting algorithm. We then show that the randomized reduction of Section 3 parallelizes very simply and derive a randomized conservative sorting algorithm.

Lemma 6.5 requires a word length of $\Theta(M^2 f)$ bits, where $M = (\log n)^{1+\Omega(1)}$, essentially because of the need to carry out all comparisons between two groups of $\log n$ keys each within a single word. We now describe how randomization allows us to solve the problem with a word length of just $O(Mf)$ bits. Since the only part of the algorithm of Lemma 6.5 that needs a word length exceeding $\Theta(Mf)$ bits is the selection subroutine of Lemma 6.2, it suffices to replace the latter with a less wasteful routine. We provide a bottom-up description and begin by considering the problem of generating a random sample. For $0 \le p \le 1$, define a *p-sample* (with respect to the (M, f)-representation) as a word representing a sequence $(b_1, \ldots, b_M)$ of independent random truth values such that $b_i = \mathit{true}$ with probability p, for $i = 1, \ldots, M$.

Lemma 7.1 *Given positive integers $M, f \ge 2$ and $h < f$ and the constant $1_{M,f}$, a 2^{-h}-sample with respect to the (M, f)-representation can be computed in constant sequential time with a word length of Mf bits.*

Proof We first use the built-in random-choice instruction to draw a random integer A from the uniform distribution over $\{0, \ldots, 2^{Mf} - 1\}$. Then all except the rightmost h bits in each field of A are masked away by executing $A := A$ AND $(1_{M,f} \cdot (2^h - 1))$, and the result is obtained as $1_{M,f} \mid [A = 0]$. $\square$

Suppose that U is a simple set stored in a word X according to the (M, f)-representation. The algorithm of Lemma 6.4 compacts U into exactly $|U|$ fields in $O(\log M)$ time. In contrast, we will now develop a randomized algorithm to compact U into roughly $|U|^3$ fields in constant time. Lemma 7.2 below handles abstract aspects of the algorithm, while Lemma 7.3 describes its implementation in our model.

The basic idea of the compaction is to multiply X by a word R containing a random subset of the 1 bits of $1_{2M,f}$ (and no other 1 bits). Observe that during the multiplication, a 1 in field i of R can be viewed as placing a copy of each element of U $i-1$ fields to the left of its position in X. Each element of U thus "multiplies" into a collection of copies, some of which may be placed in a section of consecutive fields that we choose to consider as the "target area" of the compaction. If two copies collide, in the sense that they are placed in the same field, both are lost (an unintended addition takes place); we will ensure, however, that the compaction is successful if each element of U has at least one noncolliding copy placed in the target area.

In order to facilitate the discussion, define $A+B$, where A and B are sets of integers, as $\{a+b \mid a \in A \text{ and } b \in B\}$. $A-B$ is defined analogously, and we abbreviate $\{a\}+B$ as $a+B$. In the lemma below, A represents the set of indices of fields containing elements of U. T is the set of indices of those fields that form the target area, and S corresponds to the set of indices of the nonzero fields in R; for simplicity, we allow S to contain arbitrary integers, just as we allow A and T to be arbitrary sets of integers. For $a \in A$, the set B_a defined in the lemma is easily seen to be the set of indices of fields in the target area to which the element in field a of X is copied without collision, so that the probability bounded by the lemma is the failure probability of the compaction.

Lemma 7.2 *Let $s, m \in I\!N$, let A and T be sets of integers with $|A| = s$ and $|T| = m$ and let S be a random set obtained by including each integer independently of all other integers and with some probability $p \le 1/(2s)$. For each $a \in A$, let $B_a = ((a+S) \cap T) \setminus ((A \setminus \{a\}) + S)$. Then $B_a = \emptyset$ for some $a \in A$ with probability at most $s \cdot e^{-2mp^3}$.*

PROOF It suffices to show that for arbitrary $a \in A$, $B_a = \emptyset$ with probability at most e^{-2mp^3}. Hence fix $a \in A$ and take $B = B_a$. We first investigate the probability that $a+i \notin B$, for an arbitrary integer i with $a+i \in T$. $a+i \in a+S$ with probability p, and $a+i \in (A \setminus \{a\}) + S$ with probability at most $(s-1)p \leq 1/2$. Furthermore, these two events are independent, since they depend on the inclusion in S of disjoint sets of integers, so that $a+i \notin B$ with probability at most $1 - p/2$.

We will construct a set $I \subseteq T - \{a\}$ with the property that, for each $i \in I$, we can bound the probability that $a+i \notin B$ by $1-p/2$, independently of the membership in B of any other elements of $a+I$. For this it suffices, similarly as above, that the membership in B of distinct elements of $a+I$ be determined by disjoint sets of elementary events of the form $j \in S$. For every $i \in T - \{a\}$, the event $a+i \in B$ is determined exclusively by the events $j \in S$, where $j \in \{a+i\} - A$. A sufficient condition for I having the desired property is therefore that for all distinct $i, i' \in I$, we have $(\{a+i\} - A) \cap (\{a+i'\} - A) = \emptyset$, i.e., $i' \neq i - b + b'$ for all $b, b' \in A$. Since each element of I rules out at most $s^2 - 1$ other elements of $T - \{a\}$ as potential members of I, there is a set I with the desired property of size at least $|T|/s^2 = m/s^2$. This shows that $B = \emptyset$ with probability at most $(1-p/2)^{m/s^2} \leq e^{-mp/(2s^2)} \leq e^{-2mp^3}$. □

Lemma 7.3 *Suppose that we are given integers $M, m \geq 2$ and $f \geq \log M + 2$, a word X representing a nonempty simple set U according to the (M,f)-representation, an integer $g \geq \log|U|$, and the constants $1_{M,f}$, $1_{m,mf}$, $1_{m,(m-1)f}$ and $1_{m,(m+1)f}$. Then we can compute a word representing U according to the (m,f)-representation in constant sequential time with probability at least $1 - |U| \cdot e^{-m \cdot 2^{-3g-2}}$ using a word length of $\max\{2M, m^2\} \cdot f$ bits.*

PROOF Without loss of generality assume that $m < M$ and that $g+1 < f$. Begin by removing any spurious bits by replacing X by X AND $(2^{Mf} - 1)$ and use the algorithm of Lemma 7.1 to obtain a p-sample R according to the $(2M,f)$-representation, where $p = 2^{-g-1}$. We intend to apply Lemma 7.2 with $T = \{M+1, \ldots, M+m\}$; note, in this context, that R represents only a finite section of the random set S of the lemma, but that the remaining part of S is irrelevant, since $T - \{0, \ldots, M-1\} \subseteq \{0, \ldots, 2M-1\}$ (informally, none of the missing indices could map an element of U into the target area). In accordance with the informal discussion, the bulk of the compaction is carried out by the assignment $C := ((R \cdot X) \downarrow (Mf))$ AND $(2^{mf} - 1)$ and the companion assignment $D := ((R \cdot [X > 0]) \downarrow (Mf))$ AND $(2^{mf} - 1)$; by the assumption of free leading bit positions in each field, there are no carries between fields, and C and D are valid words according to the (m, f)-representation, which we adopt (note that $1_{m,f} = 1_{M,f}$ AND $(2^{mf} - 1)$). In order to discover the fields containing noncolliding copies, we note that a field containing a noncolliding copy in C is characterized by containing the value 1 in D. Thus set $Y := C \mid [D = 1_{m,f}]$.

Y is not quite the required output, since a single element of U may be represented in Y through several (noncolliding) copies. In order to clear each copy that has a copy of the same element of U to its right, we proceed similarly as in the proof of Lemma 6.2. Assuming that $Y = (y_1, \ldots, y_m)$ and temporarily adopting the (m^2, f)-representation, we create the two words $A = Y \cdot 1_{m,mf}$ and $B = ((Y \cdot 1_{m,(m-1)f}) \mid 1_{m,mf}) \cdot 1_{m,f}$ after clearing any spurious bits in Y and set $T := [A = B] \wedge [B > 0]$. Recall that T contains the result of comparing y_i (in A) with y_j (in B), for $i = 1, \ldots, m$ and $j = 1, \ldots, m$. We clear those results that correspond to $i \leq j$ by setting $E := T \mid (\neg(1_{m,mf} \cdot 1_{m,(m+1)f}))$ (see Fig. 5) and compute $|\{j : 1 \leq j < i \text{ and } y_i = y_j\}|$, for $i = 1, \ldots, m$, by taking $F := (E \cdot 1_{m,mf}) \downarrow (m(m-1)f)$ (see Fig. 4). Finally we revert to the (m, f)-representation and return $Y \mid [F = 0]$. By Lemma 7.2, the probability that the compaction works correctly is as claimed. $\square$

The probabilistic analysis of Lemma 7.5 below is based largely on the Chernoff bounds expressed in the following lemma. For proofs, see, e.g., [23].

Lemma 7.4 *For every binomially distributed random variable Z and for all ϵ with $0 \leq \epsilon \leq 1$,*

(a) For all $z \geq E(Z)$, $\Pr(Z \geq (1 + \epsilon)z) \leq e^{-\epsilon^2 z/3}$.

(b) $\Pr(Z \leq (1 - \epsilon)E(Z)) \leq e^{-\epsilon^2 E(z)/2}$.

Lemma 7.5 *Suppose that we are given integers $M, m \geq 2$ and $f \geq \log M + 2$ with $m \leq \sqrt{M}$, but $m = \Omega(\sqrt{M})$, a word X representing a simple set U according to the (M, f)-representation, an integer r with $1 \leq r \leq |U|$, the constants $1_{M,f}$, $1_{m,mf}$, $1_{m,(m-1)f}$ and*

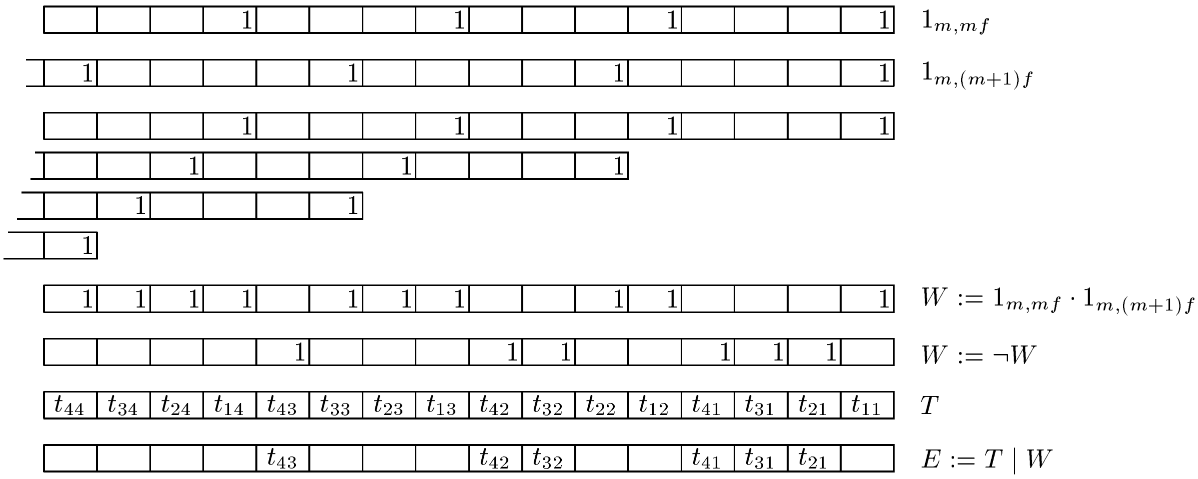

Figure 5: Removing unwanted copies.

$1_{m,(m+1)f}$ and the integer $\lceil \log |U| \rceil$. Then, for a certain constant $\epsilon > 0$, we can find the element of U whose rank in U is r in constant sequential time using a word length of $2Mf$ bits with probability at least $1 - 2^{-|U|^\epsilon}$.

Proof We use a variant of the randomized selection algorithm of Floyd and Rivest [17]. We first give a high-level overview of the algorithm, then provide more details and analyze the algorithm in an abstract setting, and finally describe its implementation in our model of computation.

In order to select the element x^* of rank r from a set U, we first draw a random sample S from U. S is chosen so small that direct selection from S is feasible; in our case, selection from S is done according to Lemma 6.2. The rank of x^* in S is likely to be close to $r \cdot |S|/|U|$. We therefore select two elements $u < v$ from S with ranks close to $r \cdot |S|/|U|$, but on opposite sides of this value, compare each element y of U with u and v and classify it accordingly as *small* $(y < u)$, *medium* $(u \leq y \leq v)$, or *large* $(y > v)$. Choosing the ranks of u and v sufficiently far from $r \cdot |S|/|U|$ ensures that with high probability x^* is medium, in which case its rank r' in the set U' of medium elements is r minus the number of small elements. The algorithm therefore discards the small and the large elements and proceeds recursively to select the element of rank r' from U'. We will ensure that U' is so much smaller than U

that the maximum depth of recursion will be bounded by a constant with high probability.

The program given below deviates from the informal outline above in certain technical aspects; e.g., recursion has been converted to iteration. The program inputs a (simple) set U, drawn from an ordered universe, and an integer r with $1 \le r \le |U|$, and it returns either the element x^* of U of rank r or a failure indication; in the latter case we will say that the execution *fails*. In accordance with the outline, the program uses an unspecified constant-time routine *Select* to perform selection from small sets. Suppose that V is a subset of U and that s is an integer. If $1 \le s \le |V|$, $\mathit{Select}(V, s)$ returns the element of V of rank s; otherwise both the call of *Select* and the program as a whole fail. In line (6), by a random p-sample of V we mean a random subset S of V obtained by including each element of V in S with probability p and independently of the inclusion of any other elements. For the sake of convenient exposition, we assume the availability of elements ∞ and $-\infty$ with $-\infty \le \min U \le \max U \le \infty$. We also assume that $h \ge 4$; if this is not the case, $|U|$ is bounded by a constant, and selecting from U can be done in constant time.

(1) $h := 2^{\lceil (\log |U|)/64 \rceil}$;
(2) $V := U$;
(3) **for** $t := 1$ **to** 57 **do**
(4)   **begin**
(5) $p := h^{t-58}$;
(6) $S :=$ Random p-sample of V;
(7)   **if** $\lceil rp \rceil - h^5 \ge 1$
(8)   **then** $u := \mathit{Select}(S, \lceil rp \rceil - h^5)$
(9)   **else** $u := -\infty$;
(10)   **if** $\lfloor rp \rfloor + h^5 \le |S|$
(11)   **then** $v := \mathit{Select}(S, \lfloor rp \rfloor + h^5)$
(12)   **else** $v := \infty$;
(13) $r := r - |\{x \in V : x < u\}|$;
(14) $V := \{x \in V : u \le x \le v\}$;
(15)   **end**;

(16) $\quad$ *return*$(Select(V, r))$;

The running time of the program is bounded by a constant, and if it does not fail, its output is correct; to see the latter, observe that if the condition $x^* \in V$ becomes false at some point during the execution, then at that point the condition $1 \le r \le |V|$ is also violated, and it remains violated until the end of the execution, where this will cause the program to fail. What remains is to bound the failure probability and the sizes of the first arguments with which *Select* is called.

For $t = 1, \ldots, 57$, let *Stage* t be the tth execution of lines (4)–(15) and denote by V_t and V_t' the value of V at the beginning and at the end of Stage t, respectively. We will say that Stage t is *successful* if the execution reaches the end of Stage t without incurring any failure, and if at that time $x^* \in V$ and $|V| \le h^{64-t}$.

Let $t \in \{1, \ldots, 57\}$ and suppose that Stage $t-1$ is successful (if $t = 1$, take the statement to be vacuously true). Then, in Stage t, the sample size $|S|$ is binomially distributed with expected value $p|V_t| \le h^{t-58} \cdot h^{64-(t-1)} = h^7$. By Lemma 7.4(a), $|S| \le 2h^7$, except with probability at most $2^{-h^7/3}$. Furthermore, since $E(|S|) = p|V_t| \ge rp$, the call of *Select* in Line (8) can fail only if $E(|S|) \ge h^5$ and $|S| \le rp - h^5 \le E(|S|)(1 - h^{-2})$. By Lemma 7.4(b), this happens with probability at most $e^{-h^5 \cdot (h^{-2})^2/2} \le 2^{-h/2}$. It is also easy to see that the call of *Select* in Line (11) cannot fail.

Let us now investigate the probability that Stage t, although not failing, is unsuccessful. If this happens, then at the end of Stage t either (1) $x^* < u$, (2) $x^* > v$, or (3) $|V| > h^{64-t}$. We consider these events in turn.

If $x^* < u$, then $Z < \lceil rp \rceil - h^5$ and hence $Z < rp - h^5$, where Z is the number of elements $\le x^*$ in V_t included in S in Stage t, and r denotes the value current at the beginning of Stage t. Since $E(Z) = rp \le h^7$, Lemma 7.4(b), used as above, shows that this happens with probability at most $2^{-h/2}$.

If $x^* > v$, then $Z \ge rp + h^5$, where Z and r are defined as above. Again since $E(Z) = rp \le h^7$, Lemma 7.4(a), used with $z = \max\{rp, h^5/2\}$, shows that this happens with probability at most $2^{-h/6}$.

If $|V| > h^{64-t}$ at the end of Stage t, some segment of h^{64-t} elements of V_t with consecutive ranks must have contributed fewer than $2h^5+1$ elements to the sample S computed in Stage t. The expected number of elements contributed to the sample by such a segment is $ph^{64-t}=h^6$. Thus, by the assumption $h \ge 4$ and Lemma 7.4(b), the probability that the actual number will be bounded by $2h^5$ is at most $2^{-h^6/8} \le 2^{-h^4}$. Since there are fewer than $|U|$ segments, the probability that $|V_t'| > h^{64-t}$ is at most $|U| \cdot 2^{-h^4}$.

The analysis above can be summed up by saying that if Stage $t-1$ is successful and $h \ge 4$, then the probability that Stage t will be unsuccessful or will call *Select* with an argument of size exceeding $2h^7$ is at most $(4+|U|) \cdot 2^{-h/6}$. Since the number of stages is constant, we can conclude that the probability that some stage will not be successful is $O(|U| \cdot 2^{-h/6})$. Finally, because the last stage being successful implies that the first argument of the call of *Select* in line (16) is of size at most h^7, we can conclude that except with probability $O(|U| \cdot 2^{-|U|^{1/64}/6})$, the complete program returns the correct result and never calls *Select* with a first argument of size exceeding $2 \cdot 2^{7\lceil(\log|U|)/64\rceil}$.

What remains is to describe the implementation of the program above. The computation of a random sample in line (6) can be done in constant time according to Lemma 7.1. The routine *Select* is implemented via the algorithm of Lemma 6.2. In order to make sure that the word length required by calls of *Select* is at most $2Mf$ bits, we first compact the first argument of each call of *Select* from the (M,f)-representation to the (m,f)-representation with the algorithm of Lemma 7.3, called with $g = 7\lceil(\log|U|)/64\rceil+1$. By assumption, $m = \Omega(M^{1/2}) = \Omega(|U|^{32/64})$, so that, for sufficiently large values of M, the probability that the compaction of a set of size at most $2 \cdot 2^{7\lceil(\log|U|)/64\rceil} = O(|U|^{7/64})$ fails is at most $2^{-|U|^{\epsilon}}$, for a suitable constant $\epsilon > 0$ (in fact, at least for any $\epsilon < 11/64$). Adding up all the failure probabilities gives a bound of the same form. $\square$

The k-halver constructed from the randomized selection procedure of Lemma 7.5 as described in the proof of Lemma 6.3 clearly will also be randomized, i.e., with a certain probability it will fail. We will employ a large number of such randomized k-halvers cooperating in a sorting network as described before the statement of Lemma 6.5, and we cannot expect all of these to operate correctly; indeed, the usual case will be that some of them fail. This raises the question of what to do with a faulty k-halver; what should it output? Our approach will be to mark the elements that at some point were output by a faulty k-halver in order to extract them later. Since their number will be small, we can afford to sort them separately with a usual sorting routine in order to merge them back into the sequence of remaining elements. This implicitly assumes, however, that the sequence of remaining elements is sorted. If a faulty k-halver simply passes on its two multiset operands without modification, this will not necessarily be the case; we will require faulty k-halvers to behave in a slightly more sophisticated way. For $k \in \mathbb{N}$, define a *k-pseudohalver* as a device that replaces two multisets U and V of k integers each either (normal operation) by $U \Cap V$ and $U \Cup V$, respectively, or (faulty operation) by the multiset consisting of k copies of $\min\{x, y\}$ and the multiset consisting of k copies of $\max\{x, y\}$, respectively, where x is an arbitrary element of U and y is an arbitrary element of V. Furthermore, define the *k-pseudohalving version* of a sorting network $\mathcal{P}$ in exactly the same way as the k-halving version of $\mathcal{P}$, but with each *Compare* instruction implemented by a k-pseudohalver, rather than by a k-halver. Note that every execution of the k-pseudohalving version of a sorting network transforms the input sequence into an output sequence of the same length as the input sequence, but one that is not necessarily a permutation of the input sequence.

The task required of a faulty k-pseudohalver was defined so as to be computationally trivial. Indeed, note that it is easy to carry out in constant (deterministic) sequential time with the algorithm of Lemma 3.1.

Lemma 7.6 *For all $k \in \mathbb{N}$ and for any sorting network $\mathcal{P}$, every output of the k-pseudohalving version $\mathcal{P}'$ of $\mathcal{P}$ is a sorted sequence.*

Proof The solution provided for [27, Exercise 5.3.4.38] happens to prove also the more general claim of Lemma 7.6. For the reader's convenience, we give a few more details.

Let m and T be the width of $\mathcal{P}$ and the number of *Compare* instructions in $\mathcal{P}$, respectively, and suppose that $\mathcal{P}'$ operates on an array $Q[1\,..\,m]$ of multisets of k elements each. Assume by way of contradiction that some execution of $\mathcal{P}'$ outputs a sequence that is not sorted and denote by $(U_1^{(t)}, \ldots, U_m^{(t)})$ the sequence of multisets stored in Q after the execution of the tth *Compare* instruction, for $t = 0, \ldots, T$ (for $t = 0$, before the execution of the first *Compare* instruction). Since the output is not sorted, there is an integer l with $2 \le l \le m$ such that some element x of $U_l^{(T)}$ is smaller than some element of $U_{l-1}^{(T)}$.

Let us call a multiset *low* if all its elements are $\le x$, and *high* if all its elements are $> x$. In the remainder of the proof, we use the term *vector* to denote a sequence of m bits. For $t = 0, \ldots, T$, we will say that a vector $(y_1, \ldots, y_m)$ is *consistent at time t* if the following holds for $i = 1, \ldots, m$: If $y_i = 0$, then $U_i^{(t)}$ is not high, and if $y_i = 1$, then $U_i^{(t)}$ is not low. By assumption, there is a vector $Y^{(T)}$ that is consistent at time T, but not sorted. For $t = T-1, \ldots, 0$, we will construct a vector $Y^{(t)}$ that is consistent at time t and with the property that if $Y^{(t)}$ is input to the program consisting of the last $T-t$ *Compare* instructions of $\mathcal{P}$, then the resulting output vector is $Y^{(T)}$. Taking $t = 0$ provides a vector $Y^{(0)}$ that fails to be sorted by $\mathcal{P}$, a contradiction.

Let $t \in \{0, \ldots, T-1\}$ and assume that we have already constructed $Y^{(t+1)} = (y_1, \ldots, y_m)$ with the desired properties and that the $(t+1)$st *Compare* instruction of $\mathcal{P}$ is *Compare*(i,j). We take $Y^{(t)}$ to agree with $Y^{(t+1)}$ in all components except the ith and the jth, i.e., $Y^{(t)} = (y_1, \ldots, y_{i-1}, y'_i, y_{i+1}, \ldots, y_{j-1}, y'_j, y_{j+1}, \ldots, y_m)$, where y'_i and y'_j will be chosen below. Provided that $\{y'_i, y'_j\} = \{y_i, y_j\}$, it is clear that the $(t+1)$st *Compare* instruction of $\mathcal{P}$ transforms $Y^{(t)}$ to $Y^{(t+1)}$, and hence that the last $T-t$ *Compare* instructions of $\mathcal{P}$ transform $Y^{(t)}$ to $Y^{(T)}$, as required.

The number of low (high, respectively) multisets in $\{U_i^{(t)}, U_j^{(t)}\}$ is no larger than the number of low (high) multisets in $\{U_i^{(t+1)}, U_j^{(t+1)}\}$, independently of whether the operation of the k-pseudohalver implementing the $(t+1)$st *Compare* instruction of $\mathcal{P}$ is normal or faulty. This observation shows that it is always possible to choose y'_i and y'_j with $\{y'_i, y'_j\} = \{y_i, y_j\}$ such that $Y^{(t)}$ is consistent at time t. E.g., if $U_i^{(t)}$ and $U_j^{(t)}$ are both low, then $U_i^{(t+1)}$ and $U_j^{(t+1)}$ are also both low, $y_i = y_j = 0$ by consistency, and we can take $y'_i = y'_j = 0$. $\square$

Lemma 7.7 *For all given integers $n, q \ge 4$ and $b \ge \log\log n$ and all fixed $\delta > 0$, n integers of b bits each can be value-sorted in $O(\log n(1 + \log\log n/\log q))$ time using $O(n(1 + \log\log n/q^{1-\delta}))$ operations on a unit-cost EREW PRAM with a word length of $O(bq\log n)$ bits and the full instruction set with probability at least $1 - 2^{-n/(\log n)^3}$.*

PROOF Without loss of generality, we assume that $q \leq \log n$ and that $\delta \leq 1$ and choose k as the smallest power of 2 no smaller than $\lfloor q^{1-\delta} \rfloor \cdot \lfloor \log n \rfloor$. We essentially carry out the construction of Lemma 6.5, but replace each k-halver by a k-pseudohalver implemented according to Lemma 7.5, modified so that in the event of a faulty operation, it will save the two words constituting its input and mark each of the $2k$ elements in its output. One additional (rightmost) bit must be set aside in each field for the mark; the resulting increase in the field width is insignificant, however. After executing the procedure of Lemma 6.5, modified as just described, we remove all marked elements from the output sequence and call the sequence of remaining elements S; by Lemma 7.6, S is sorted. We also remove all marked elements from the set of elements saved by faulty k-pseudohalvers (a marked element in this set was output by an earlier faulty k-pseudohalver) and call the resulting set F. We finally sort F using the algorithm of Cole [12] and merge the resulting sorted sequence with S using one of the algorithms of [10, 22] to obtain the sorted output sequence.

The procedure works in $O(\log n)$ time plus the time needed for compaction according to Lemma 6.4. Since $b \geq \log \log n$, a field length f of $O(b)$ bits suffices, so that a word length of $\Theta(bq \log n)$ and the choice of $k = \Theta(q^{1-\delta} \log n)$ allows a "blowup" of $\Theta(q^{\delta})$ between compactions. We choose to compact at every dth level of the AKS network, where $d \leq \delta \log q$, but $d = \Omega(\log q)$. This takes $O(\log \log n)$ time per compaction, and $O(\log n(1 + \log \log n / \log q))$ time overall. The number of operations executed depends on the number of faulty k-pseudohalvers and is $O(n(1 + \log \log n/(q^{1-\delta} \log q)) + |F| \log |F|)$. Since $|U| = 2k \geq \log n$ in the application of Lemma 7.5, the expected number of faulty k-pseudohalvers is $O(n/(\log n)^3)$. Hence, by Lemma 7.4(a), the actual number of faulty k-pseudohalvers is $O(n/(\log n)^3)$ with probability at least $1 - 2^{-n/(\log n)^3}$. Since each faulty k-pseudohalver contributes at most $2k = O((\log n)^2)$ elements to F, $|F| = O(n/\log n)$ and the number of operations executed is $O(n(1 + \log \log n/q^{1-\delta}))$ with the same probability. $\square$

In the interest of simplicity, Lemma 7.7 above restricts the quantity δ to be a constant. A more general form of the lemma would allow smaller values of δ and exhibit a trade-off between time and work. The same is true of Theorem 7.1 below.

Theorem 7.1 *For all given integers $n \geq 4$ and $w \geq 2(\log n)^2$ and all fixed $\delta > 0$, a unit-cost*

EREW PRAM with a word length of w bits and the full instruction set can sort n integers in the range $0\,..\,2^w - 1$ in $O(\log n(1+\log\log n/\log q)^2)$ time using $O(n(1+\log\log n/\log q)(1+\log\log n/q^{1-\delta}))$ operations, where $q = w/(\log n)^2$, with probability at least $1 - 1/n$.

PROOF Recall that the major steps in the randomized signature-based range reduction of Section 3 were to compute the concatenated signatures of the input keys, to sort these, then to construct their compressed trie T_{D}, and finally to sort the children of each internal node in T_{D} not by the relevant signatures, but instead by the corresponding original fields.

The sequential computation of the concatenated signatures of the input keys parallelizes trivially, since it is done independently for each key. The same is true of the computation of the lengths $r_1, \ldots, r_{n-1}$ of the longest common prefixes of consecutive concatenated signatures. The Cartesian tree T_{C} of $(r_1, 1), \ldots, (r_{n-1}, n-1)$ can be constructed in $O(\log n)$ time using $O(n)$ operations on an EREW PRAM, as observed in [21], and the compressed trie T_{D} can be derived from T_{C} by using the Euler-tour technique [34] and optimal list ranking [14, 3] to provide each internal node in T_{D} with an array specifying its children and a pointer to the leftmost leaf, say, in its subtree. The same method allows us to collect the leaves of T_{D} in left-to-right order after the sorting at the internal nodes, which concludes the sorting.

If $w \le \lfloor \log n \rfloor^4$, we proceed similarly as in the proof of Theorem 3.1, but substituting a value of $\Theta(q^{\delta/2})$ for q. This increases the necessary number of reduction steps by a constant factor, but allows us to sort concatenated signatures using $O(\log n(1+\log\log n/\log q))$ time and $O(n(1+\log\log n/q^{1-\delta}))$ operations in each of the $O(1+\log\log n/\log q)$ reduction steps by means of the algorithm of Lemma 7.7. If $w > \lfloor \log n \rfloor^4$, we can easily use Lemma 7.7 with $q = \Theta(\log n)$ to sort in a constant number of reduction steps, each of which needs $O(\log n)$ time and $O(n)$ operations. We omit the details, which are similar to those of the proof of Theorem 3.1. □

Corollary 7.1 *If $w \ge (\log n)^{2+\epsilon}$ for some fixed $\epsilon > 0$, we can sort n integers in $O(\log n)$ expected time on an EREW PRAM using $O(n)$ expected operations.*

8 Sorting multiple-precision integers in parallel

In this section we show that an algorithm of [5] can be parallelized to yield an algorithm that reduces the problem of sorting n strings of characters from the alphabet $\{0, \ldots, m-1\}$ of total length N and maximum individual length L to that of sorting n integers in the range $1 \,..\, L$ and, subsequently, at most $2n$ integers in the range $0 \,..\, nm-1$. The reduction uses $O(L + \log n)$ time and $O(N + n)$ operations on a CRCW PRAM.

We associate an array of size m with each input string. The algorithm partitions the input strings into ever smaller groups in L successive stages. After Stage t, for $t = 1, \ldots, L$, two strings are in the same group if and only if they have the same prefix of length t. Furthermore, each group has a *leader*, a distinguished string in the group whose index is known to all members of the group. In Stage $t+1$, each group, say, of those strings with a common prefix s of length t, collectively determines whether all its members have the same $(t+1)$st character, which is easy to do in constant time using concurrent writing to a cell in the array of the leader of the group. If this is the case, the group remains unchanged and with the same leader. Otherwise, all processors in the group with a $(t+1)$st character of a compete to write their indices to position a in the array of the leader of the group; whoever wins becomes the leader of the group containing the strings beginning with s, followed by a, which is said to have a as its *distinguishing character*.

After Stage L, the set of all groups created in the process forms a conceptual tree: The parent of each nonroot group is the smallest superset group in the set, from which it was formed directly. Furthermore, the children of each group are ordered from left to right by the numerical order among their distinguishing characters. This tree is the path-compressed trie T of the input strings. In order to actually construct the tree, each leader of a group generates the triple (i, a, j), where i is the index of the leader of the parent group (which can easily be remembered), a is the distinguishing character of the group, and j is its own index; note that a string may be the leader of several groups, in which case it will generate several triples. The triples generated can be identified with the edges of T in a natural way. Sorting the at most $2n$ triples lexicographically by their two first components brings together all edges from a node to its children and thus constructs T, modulo trivial details of the representation. The input strings now appear in sorted order at the leaves of T and, as in Section 7, can be collected via an application of the Euler-tour technique [34], followed by optimal list ranking [14, 3].

The description above implicitly associated a processor with each string; but after t stages, the processor associated with a string of length t is no longer needed. If we initially sort the strings by their lengths, it is an easy matter to simulate virtual processors, one associated with each string, by a smaller number of physical processors, in such a way that the total number of operations spent becomes $O(N + n)$.

Combining the reduction with Theorem 6.1 and Corollary 7.1 and sorting integers in the range $1 \,..\, L$ using an algorithm of [13, 35], we obtain the following result.

Theorem 8.1 *For all integers $n, L, N \geq 4$, n multiple-precision integers occupying at most L machine words each and N machine words altogether can be sorted either in $O(L + \log n)$ time using $O(N + n \log\log n)$ operations on a CRCW PRAM or, provided that $w \geq (\log n)^{2+\epsilon}$ for some fixed $\epsilon > 0$, in $O(L + \log n)$ expected time using $O(N + n)$ expected operations on a randomized CRCW PRAM.*

9 Space requirements of parallel sorting

In this section we show that the additional use of randomization and division allows the second algorithm of Theorem 6.1 to be implemented in linear space. The only part of the deterministic algorithm with superlinear space requirements is the solution of ordered-chaining problems using the reduction algorithm of Bhatt et al. [9]; we therefore need to take a closer look at the latter.

Recall that the goal is to reduce an ordered-chaining problem of size N, where N is a power of 2, to $2^{\lceil (\log N)/2 \rceil} + 1$ ordered-chaining problems, each of size at most $2^{\lceil (\log N)/2 \rceil}$. The reduction works as follows: The processors, inactive as well as active, are partitioned into $2^{\lceil (\log N)/2 \rceil}$ groups of $2^{\lfloor (\log N)/2 \rfloor}$ consecutively numbered processors each, each of which defines a *local* ordered-chaining problem; for $i = 1, \ldots, 2^{\lceil (\log N)/2 \rceil}$, the ith group consists of the processors numbered $(i-1) \cdot 2^{\lfloor (\log N)/2 \rfloor}, \ldots, i \cdot 2^{\lfloor (\log N)/2 \rfloor} - 1$. Call a group *nonempty* if it contains at least one active processor. Each nonempty group uses concurrent writing to elect one of its active processors as a *leader*, and the leaders thus contributed by the nonempty groups leave their original groups, changing the original local ordered-chaining problems into *modified* local problems, and form a *global* ordered-chaining problem. At this point the modified local problems and the global problem are solved recursively using the same method, after which constant time suffices to re-insert the leaders, i.e., to obtain solutions to the original local problems, and to link the nonempty subproblems together correctly by means of the solution to the global problem.

Note the structure of the reduction algorithm: Some constant-time initial computation is followed by parallel recursive calls, which in turn are followed by constant-time final computation. Exclusive of the needs of recursive calls, a subproblem of size m arising in the process requires at most cm memory cells for its computation, for some constant c. Bhatt et al. analyze the corresponding recurrence equation and show that when recursive invocations are taken into account, it suffices to provide a subproblem of size $m \geq 2$ with $c_1 m - c_2$ cells, for some other constants c_1 and c_2. This makes it a simple matter to allocate space for recursive calls during the initial computation, and at each recursive level each processor can determine the memory area of the subproblem to which it will be assigned at the next recursive level in constant time.

The actual memory requirements of the initial computation are a single cell per subproblem, used to elect the leader. Since there are only n active processors, the number of cells written to in a single step never exceeds n. The at most n addresses used in a particular step may come from a large domain. Using the hashing scheme of Bast and Hagerup [6], however, they can be mapped down to integers in a range of size $O(n)$, so that the election of leaders can be done using $O(n)$ space. This needs $O(\log^* n)$ time and $O(n)$ operations, which add up to negligible quantities over the $O(\log\log n)$ recursive stages, and incurs a failure probability of $2^{-n^{\Omega(1)}}$. Furthermore, the values written are never used again, so that the same $O(n)$ cells can be reused over all recursive stages.

The same approach can be used to deal with the memory requirements of the final computation, except that we must describe how a processor that proceeds to the final computation can find the corresponding memory area (say, its first cell). Storing this information on a stack from the corresponding initial computation in a straightforward way would work, but the stacks would together take up $\Theta(n \log \log n)$ space. Instead we store only the *displacement* relative to the memory area of the previous recursive level. It can easily be arranged that the size of these displacements decreases rapidly with the recursive depth (indeed, the most natural memory layout has this property), so that all displacements registered by a processor fit in a constant number of words, which can still support stack operations in constant time.

10 Conclusions

The comparison-based model is an elegant and general framework in which to study sorting problems, and the $\Theta(n \log n)$ complexity of sorting is one of the basic tenets of computer science. However, many sorting problems of considerable interest can be cast as integer-sorting problems. The complexity of integer sorting on RAM-like models therefore is of great practical and theoretical significance.

The problem of integer sorting is sometimes equated with that of sorting n integers of $O(\log n)$ bits each, another classical and well-understood problem, solved using indirect addressing in the form of radix sorting. However, it seems more natural to tie the size of the integers to be sorted not to the input size, but to the word length of the computer on which the sorting problem arises. A fundamental question therefore is: How fast can we sort n w-bit integers on a w-bit machine? Fredman and Willard achieved a breakthrough by showing the complexity to be $o(n \log n)$, independently of w. In a practical vein, they suggested that the use of features found on typical machines other than indirect addressing and comparison might eventually lead to new sorting schemes with the potential of outperforming both comparison-based sorting and radix sorting in certain settings.

The actual algorithm proposed by Fredman and Willard probably is impractical. Our sequential algorithms are simpler, have smaller constant factors, require much shorter word lengths to be effective and offer greater improvements over comparison-based sorting. Moreover, like the algorithm of Fredman and Willard, they do not rely on exotic instructions (indeed, the deterministic algorithm eschews even the use of multiplication). Nevertheless, several factors remain that probably preclude them from being practical. For instance, the deterministic algorithm has inordinate storage requirements, a property that it inherits from the algorithm of Kirkpatrick and Reisch, and both algorithms still rely on word sizes much larger than those commonly found in current machines. In the case of the deterministic algorithm, the last claim can be partially countered by observing that the exclusive use of AC^0 instructions could make the unit-cost assumption remain valid even for fairly large word sizes. Still, our results are best viewed as no more than a step further towards the goal of faster practical integer-sorting algorithms.

Our research also raises a number of intriguing theoretical questions. One is to find tight bounds on deterministic integer sorting. Can the performance of signature sort be matched by a deterministic algorithm? And can integers be sorted in linear expected time for all word lengths? We have demonstrated that n integers can be sorted in $O(n)$ expected time with a word length of w bits not only for $w = O(\log n)$, but also for $w \geq (\log n)^{2+\epsilon}$, for arbitrary fixed $\epsilon > 0$. Between these two outer ranges, however, there might be a "hump", where the complexity of integer sorting goes up to $\Theta(n \log\log n)$. We leave as an open problem to demonstrate the presence or the absence of such a "hump".

References

1. M. Ajtai, J. Komlós, and E. Szemerédi, An $O(n \log n)$ sorting network, *in* "Proc. 15th Annual ACM Symp. on Theory of Computing, 1983," pp. 1–9.

2. S. Albers and T. Hagerup, Improved parallel integer sorting without concurrent writing, *in* "Proc. 3rd Annual ACM-SIAM Symp. on Discrete Algorithms, 1992," pp. 463–472.

3. R. J. Anderson and G. L. Miller, Deterministic parallel list ranking, *in* "Proc. 3rd Aegean Workshop on Computing, 1988," Vol. 319, Springer-Verlag Lecture Notes in Computer Science, pp. 81–90.

4. A. Andersson, Faster deterministic sorting and searching in linear space, *in* "Proc. 37th Annual IEEE Symp. on Foundations of Computer Science, 1996," to appear.

5. A. Andersson and S. Nilsson, A new efficient radix sort, *in* "Proc. 35th Annual IEEE Symp. on Foundations of Computer Science, 1994," pp. 714–721.

6. H. Bast and T. Hagerup, Fast and reliable parallel hashing, Manuscript. A preliminary version appeared in "Proc. 3rd Annual ACM Symp. on Parallel Algorithms and Architectures, 1991," pp. 50–61.

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

8. A. M. Ben-Amram and Z. Galil, When can we sort in $o(n \log n)$ time? *in* "Proc. 34th Annual IEEE Symp. on Foundations of Computer Science, 1993," pp. 538–546.

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

10. G. Bilardi and A. Nicolau, Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines, *SIAM J. Comput.* **18** (1989), 216–228.

11. J. L. Carter and M. N. Wegman, Universal classes of hash functions, *J. Comput. System Sci.* **18** (1979), 143–154.

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

13. R. Cole and U. Vishkin, Deterministic coin tossing with applications to optimal parallel list ranking, *Inform. and Control* **70** (1986), 32–53.

14. R. Cole and U. Vishkin, Approximate parallel scheduling. Part I: The basic technique with applications to optimal parallel list ranking in logarithmic time, *SIAM J. Comput.* **17** (1988), 128–142.

15. M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen, "A reliable randomized algorithm for the closest-pair problem," Tech. Rep. no. 513, Fachbereich Informatik, Universität Dortmund, 1993.

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

17. R. W. Floyd and R. L. Rivest, Expected time bounds for selection, *Comm. ACM* **18** (1975), 165–172.

18. M. L. Fredman and D. E. Willard, Surpassing the information theoretic bound with fusion trees, *J. Comput. System Sci.* **47** (1993), 424–436.

19. O. Gabber and Z. Galil, Explicit constructions of linear-sized superconcentrators, *J. Comput. System Sci.* **22** (1981), 407–420.

20. H. N. Gabow, J. L. Bentley, and R. E. Tarjan, Scaling and related techniques for geometry problems, *in* "Proc. 16th Annual ACM Symp. on Theory of Computing, 1984," pp. 135–143.

21. T. Hagerup, Optimal parallel string algorithms: Merging, sorting and computing the minimum, *in* "Proc. 26th Annual ACM Symp. on the Theory of Computing, 1994," pp. 382–391.

22. T. Hagerup and C. Rüb, Optimal merging and sorting on the EREW PRAM, *Inform. Process. Lett.* **33** (1989), 181–185.

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

24. T. Hagerup and H. Shen, Improved nonconservative sequential and parallel integer sorting, *Inform. Process. Lett.* **36** (1990), 57–63.

25. J. L. Hennessy and D. A. Patterson, "Computer Organization and Design: The Hardware/Software Interface," Morgan Kaufmann Publ., San Mateo, CA, 1994.

26. D. Kirkpatrick and S. Reisch, Upper bounds for sorting integers on random access machines, *Theoret. Comput. Sci.* **28** (1984), 263–276.

27. D. E. Knuth, "The Art of Computer Programming, Vol. 3: Sorting and Searching," Addison-Wesley, Reading, MA, 1973.

28. C. P. Kruskal, Searching, merging, and sorting in parallel computation, *IEEE Trans. Comput.* **32** (1983), 942–946.

29. F. T. Leighton, "Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes," Morgan Kaufmann Publ., San Mateo, CA, 1992.

30. R. Paige and R. E. Tarjan, Three partition refinement algorithms, *SIAM J. Comput.* **16** (1987), 973–989.

31. W. J. Paul and J. Simon, Decision trees and random access machines, *in* "Proc. International Symp. on Logic and Algorithmic, Zürich, 1980," pp. 331–340.

32. N. Pippenger, Communication networks, *in* J. van Leeuwen, ed., "Handbook of Theoretical Computer Science, Vol. A, Algorithms and Complexity," Elsevier, Amsterdam/The MIT Press, Cambridge, MA, 1990.

33. R. Raman, Priority queues: small, monotone and trans-dichotomous, *in* "Proc. 4th Annual European Symp. on Algorithms, 1996," to appear.

34. R. E. Tarjan and U. Vishkin, An efficient parallel biconnectivity algorithm, *SIAM J. Comput.* **14** (1985), 862–874.

35. R. A. Wagner and Y. Han, Parallel algorithms for bucket sorting and the data dependent prefix problem, *in* "Proc. International Conf. on Parallel Processing, 1986," pp. 924–930.