Sorting in Linear Time?

Arne Andersson* Torben Hagerup† Stefan Nilsson* Rajeev Raman‡

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 $O(n \log n)$ 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 are 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 [13, p. 228]. Most 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. Bucket sorting requires the n input keys to be in the range $0\,..\,n-1$. 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 [14], reduces this to $O(n \log k)$, 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, we can actually sort integers in linear time independently of their size, as demonstrated by Paul and Simon [18] and Kirkpatrick and Reisch [14]. 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 it is realistic to assume that a full repertoire of "reasonable" instructions can be applied to word-sized operands in constant time. 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.

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

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

STOC '95, Las Vegas, Nevada, USA

Fredman and Willard [9] were the first to show that n arbitrary integers can be sorted in $o(n \log n)$ time by a conservative method. Their algorithm, based on *fusion trees*, sorts n integers in $O(n\sqrt{\log n})$ time. We describe two simple algorithms that improve their result. It should be noted that fusion trees have other uses besides sorting, such as in efficient data structures, to which our results do not apply.

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 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 [4], 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 [3]). 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 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.

Ben-Amram and Galil [5, Theorem 5] have shown that, under 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. We also obtain 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, which is worst-case optimal if $N = \Omega(n \log\log n)$.

Our results flow from the combination of the two techniques of *packed sorting* and *range reduction*. Packed sorting, introduced by Paul and Simon [18] and developed further in [12] 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, 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.

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 radix sorting works in linear time and space). Also, standard algorithms for multiple-precision arithmetic allow us to assume constant-time operations on words of $O(w)$ bits, rather than exactly w bits.

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

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

Theorem 1 *For all given integers $n \geq 4$ and $w \geq \log n$, n integers in the range $0\,..\,2^w - 1$ 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 $O(n \log n \log k/k) = O(n)$.

Our second ingredient is the range reduction of Kirkpatrick and Reisch [14, 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 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 a new *signature sort* algorithm and show that it works in linear expected time. Signature sort is obtained by combining the packed sorting of [2] with a new, randomized range-reduction scheme.

We first provide an informal sketch of the main ideas behind signature sort. The algorithm uses packed sorting twice: to construct a path-compressed trie of hash codes and to sort the edges of this trie. Assume a word length of w bits and a parameter $k < w$ chosen to allow linear-time packed sorting of k-bit integers. In order to sort n w-bit keys, we split each key into w/k fields of k 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. The signatures of all fields in a key can be computed together in constant time, and their concatenation occupies just $O(w \log n/k)$ bits. If k is sufficiently large, the concatenated signatures of the input keys can be sorted in linear time, after which we construct their path-compressed trie (with signatures considered as characters) in linear 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. After sorting the "sibling" edges below each node in the tree by the original field values corresponding to their distinguishing signatures, the sorted sequence of the n input keys can be read off the tree in a left-to-right scan.

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*.

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 the *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 $\tau(0) = \mathit{false}$.

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 $\lfloor x \cdot 2^i \rfloor$, 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 $1_{2M,f} = 1_{M,f} \cdot (1 + 2^{Mf})$. As will be seen below, 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 $i = 1,\ldots,M$, $y_i = x_i$ if $b_i = \mathit{true}$, while $y_i = 0$ if $b_i = \mathit{false}$. $X \mid B$ can be obtained simply as X AND $(B \cdot (2^f - 1))$.

Lemma 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 $f \geq \log M + 2$ 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 give a high-level description of the new range reduction that ignores details such as rounding. Assume that $w \geq 2(\log n)^2 \log\log n$.

In order to sort n keys of b bits each, we begin by conceptually partitioning each key into b/k k-bit fields, where k is chosen so that we can still just sort k-bit integers in linear time; we know from Section 2 that we should choose $k = \Theta(w/(\log n \log\log n))$. Assume that we are given a function $h : \{0, \ldots, 2^k - 1\} \to \{0, \ldots, 2^l - 1\}$, where $l < k$, 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 $x_1, \ldots, x_{b/k}$, we define the *concatenated signature* of X as the integer obtained by concatenating (the f-bit strings representing) the signatures $h(x_1), \ldots, h(x_{b/k})$.

We now sort the n input keys by their concatenated signatures. This is the sorting problem to which we reduce the original problem; because $l < k$, it will be easier than the original problem—the fields have "shrunk". 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., $Y_1 \leq Y_2 \leq \cdots \leq Y_n$) and take $\mathcal{Y} = \{Y_1, \ldots, Y_n\}$, formed as a multiset. Viewing the elements of $\mathcal{Y}$ as character strings of length b/k 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 [11].) 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. 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 1, applied to words of the form $[Y_i = Y_{i+1}]$, this can be done in a total time of $O(n)$. Guided by this information, we can construct T_D in $O(n)$ time by means of an algorithm of Gabow et al. [10] for constructing a so-called *Cartesian tree*, which is closely related to T_D [11].

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 single 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 at most $2n$ fields within disjoint groups. We have taken care to ensure that fields are small enough to be sorted in linear time, so that the sorting at the internal nodes in T_D can be done in $O(n)$ time altogether. All that is required to finish the sorting 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 [3].

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 $O(n \log n \log\log n)$ fields. While it appears difficult to ensure this deterministically, it turns out that if l is sufficiently large, a function chosen at random from a suitable class of hash functions is injective on S with high probability. 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. This runs counter to our purpose, and it is necessary to obtain a smaller integer by packing the signatures tightly in adjacent f-bit fields. Fig. 1 shows how this can be done with a simple extension of the algorithm of [9, Lemma 3]. We pack fields

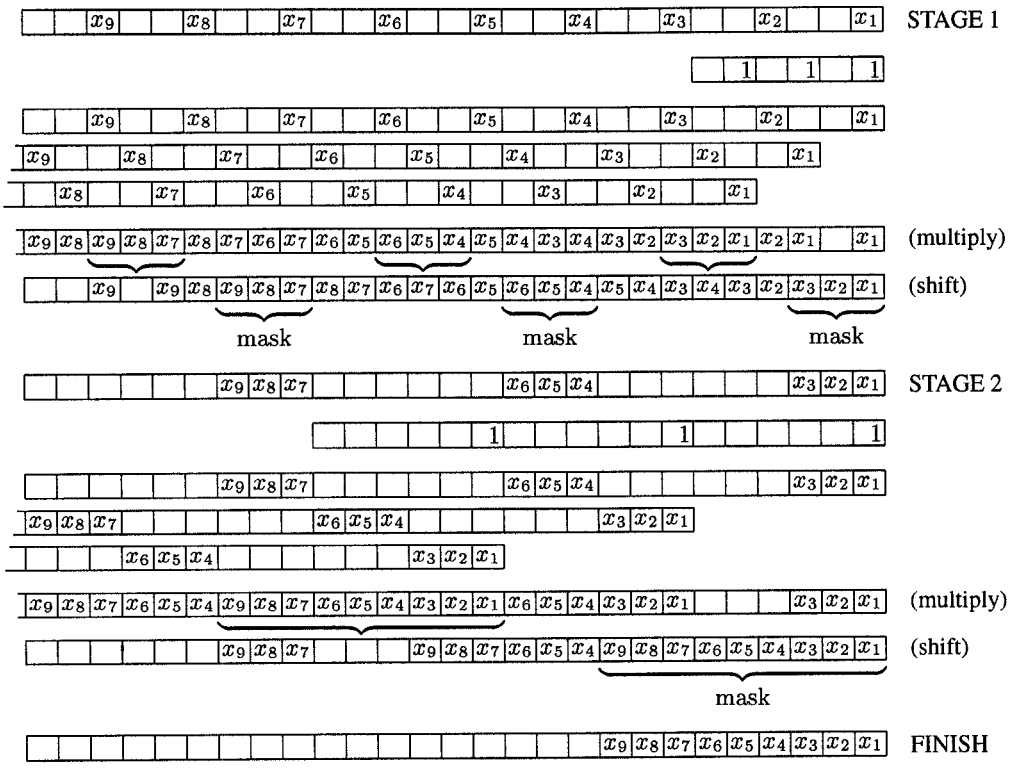

Figure 1: Packing fields tightly.

tightly in ever larger groups, starting from a group size of 1. With one multiplication, a shift and a masking operation, we can increase the size of all groups by a factor of k/f. Since the total number of fields is b/k, the complete packing takes time $O(1 + \log(b/k)/\log(k/f)) = O(1 + \log b/\log(k/f))$.

The class $\mathcal{H}$ was analyzed by Dietzfelbinger et al. [8], 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 $1 - |S|^2/2^l$. Since $|S| = O(n \log n \log\log n)$, 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 reduction. The original problem is reduced in $O(n(1 + \log b/\log(k/f)))$ time to that of sorting n concatenated signatures, each of which is a factor of k/f smaller than the original input keys. By our choice of parameters, $k/f = \Theta(q)$, where $q = w/((\log n)^2 \log\log n) \geq 2$, and we are free in our choice of k to ensure that in fact $k/f \geq q$. For $n \geq 1$, $1 \leq b \leq w$ and $p > 0$, denote by $T(n, b, p)$ the time needed to sort n integers of b bits each with probability at least p, assuming b and w to be known. The reduction can be summarized in the recurrence relation

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

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. Furthermore, $\log b/\log q = O(1 + \log\log n/\log q)$ for $b \leq w$. This proves the following main result.

Theorem 2 *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*

$$O(n(1 + \log\log n/\log q)^2)$$

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

Corollary 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

The *forward radix sort* of Andersson and Nilsson [3] 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 itself needs $O(N + n)$ time. Combining this with Theorem 1 and Corollary 1, we obtain two algorithms for the general lexicographic sorting problem.

Corollary 2 *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 [14], 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. Each reduction step pushes a list of n numbers on the stack. However, the number of bits needed to store these numbers is reduced by a factor of essentially two from one reduction step to the next. Hence, by storing several numbers in each machine word, we can arrange that the total space requirements for the recursion stack are $O(\sum_{i=0}^{\infty} n/2^i) = O(n) = O(2^w)$.

By breaking each input key into r pieces of at most $\lceil w/r \rceil$ bits each, for some $r \geq 1$, 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 2 itself works in $O(n + 2^{w/r})$ space.)

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

6 Parallel sorting

We begin this section by discussing two deterministic parallel packed-sorting algorithms. We then show how to parallelize the range reduction of Kirkpatrick and Reisch and use this to obtain a deterministic conservative parallel sorting algorithm (Theorem 4). Subsequently we argue that the range reduction of signature sort parallelizes in a straightforward manner and derive a randomized conservative parallel sorting algorithm (Theorem 5). Finally we consider the problem of sorting multiple-precision integers in parallel.

Lemma 2 *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 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 [16]. □

The algorithms of Lemma 2 need more than logarithmic time because they are based on repeated merging. We now provide an alternative algorithm that 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 3 *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; 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 [18], is to create words A and B such that field number $(i-1)M + j$ of A contains x_j, while the corresponding field of B contains x_i, 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 $[A \geq B]$. A is easily computed as $X \cdot 1_{M,Mf}$, and B can be obtained as $((X \cdot 1_{M,(M-1)f}) \mid 1_{M,Mf}) \cdot 1_{M,f}$.

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. 2). 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 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 $(2^f - 1)$. □

Given two multisets U and V of integers containing the same number k of elements, we denote by $U \hat{\wedge} V$ and $U \hat{\vee} V$ 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 $U \hat{\wedge} V$ and

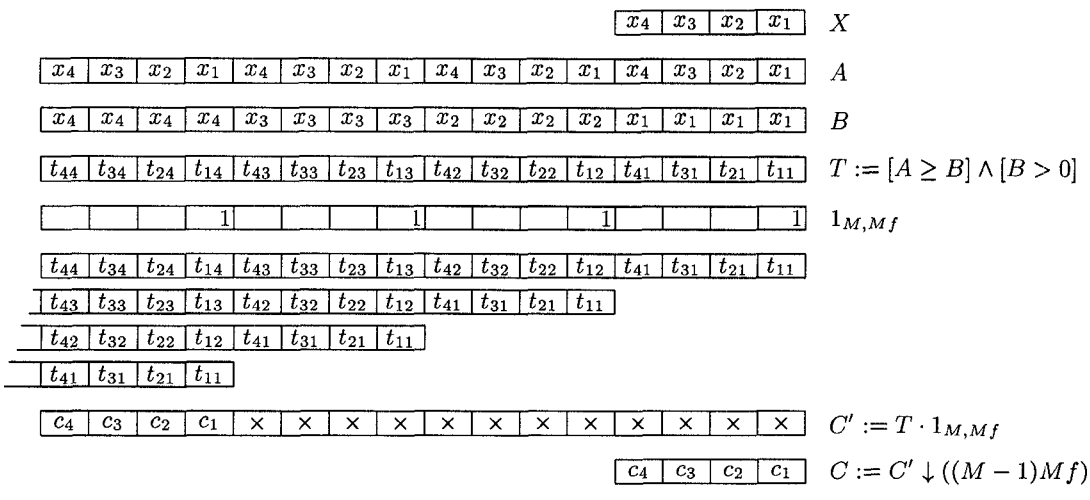

Figure 2: Computing the ranks of the x_i's. t_{ij} denotes the result of the comparison $x_i \stackrel{?}{\geq} x_j$, for $i = 1, \ldots, M$ and $j = 1, \ldots, M$, and c_i is the rank of x_i, for $i = 1, \ldots, M$. $\times$ represents a don't-care value.

$U \Vee V$. The following lemma describes the implementation of a k-halver.

Lemma 4 *Suppose that we are given integers $M \geq 2$, $m = \lceil \log M \rceil + 2$, and $f \geq m + 1$, two words X and Y representing multisets U and V of the same cardinality k according to the (M, f)-representation, and the constants $1_{M,f}$, $1_{2M,2Mf}$, and $1_{2M,2(M-1)f}$. 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 $U \Wedge V$ and $U \Vee V$ 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 3. Before we can appeal to Lemma 3, 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 3 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. □

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 5 *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 $(|U|, f)$-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 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., [17, 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 $(((X \mid A) \downarrow (2^t f)) \text{ AND } (2^{Mf} - 1)) + (X \mid (\neg A))$, for $t = 0, \ldots, \lceil \log M \rceil - 1$. □

For our purposes, a *comparator network* of *width* m is a straight-line program consisting of a sequence of instructions of the form *Compare*(i, j), where $1 \leq i < j \leq m$. The intended semantics is that a comparator network of width m operates on an array $Q[1..m]$ containing m (not necessarily distinct) elements drawn from an ordered universe, and that the execution of an instruction *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 $Q[1] \leq Q[2] \leq \cdots \leq Q[m]$ 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 $m \geq 2$, 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 reinterpret 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 *Compare*(i, j) simultaneously replaces $Q[i]$ and $Q[j]$ by $Q[i] \wedge Q[j]$ and $Q[i] \vee Q[j]$, 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 correctly sorts any sequence of km elements [15, Exercise 5.3.4.38].

Theorem 3 *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 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 no smaller 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 4. 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 algorithm of Lemma 2. 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 4, 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 5 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 $M_{\max}^2 f$ bits (Lemma 4 is the bottleneck). By the discussion above, this is $O(f(\log n)^{2+\epsilon/2})$ bits. According to Lemma 4, 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 4 in $O(\log n)$ sequential time for all relevant values of M. Finally, although we shall not demonstrate it here, the AKS network can be constructed within the required resource bounds. □

The range reduction of Kirkpatrick and Reisch does not lend itself to easy direct parallelization. Bhatt et al. [6] 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. The ordered-chaining problem of size N is, given N processors numbered $0, \ldots, N-1$, some of which are (permanently) inactive, to compute for each active processor the smallest processor

number of an active processor larger than its own number, if any; the active processors are thus to be hooked together in a linked list.

The problem of sorting n keys in the range $0..m-1$ can also be viewed as that of sorting n *distinct* integers in the range $0..nm-1$, which can in turn be viewed as an ordered-chaining problem of size $N = nm$. Bhatt et al. describe a reduction that takes constant time on a CRCW PRAM and essentially reduces an ordered-chaining problem of size N to a collection of ordered-chaining subproblems of size $\sqrt{N}$ each. Our approach is to apply this reduction r times, for suitable r, and to solve the resulting ordered-chaining subproblems by viewing them as integer-sorting problems (sort the processors within each subproblem by their processor numbers, which are integers of length smaller than the input keys by a factor of 2^r). Undoing the reductions yields a solution to the original ordered-chaining problem, which is turned into a solution to the original integer-sorting problem by means of optimal list ranking [7]. The whole reduction needs $O(r + \log n)$ time and $O(nr)$ operations.

Theorem 4 *For all given integers $n \geq 4$ and $w \geq \log n$, n integers in the range $0..2^w - 1$ can be sorted using $O(n \log\log n)$ operations in $O(\log n \log\log n)$ time on a unit-cost CRCW PRAM with the restricted instruction set, or in $O(\log n)$ time on a unit-cost CRCW PRAM with the full instruction set.*

PROOF We apply the reduction above with $r = 3\lceil \log\log n \rceil$, which takes $O(\log n)$ 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 [6] if $w \leq (\log n)^4$, and using the packed-sorting algorithm either of the second part of Lemma 2 or of Theorem 3 otherwise. □

Theorem 5 *For all given integers $n \geq 4$ and $w \geq 2(\log n)^2$, 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 $O(n(1 + \log\log n/\log q)^2)$ operations, where $q = w/(\log n)^2$, with probability at least $1 - 1/n$.*

PROOF We will demonstrate the theorem only for $w \geq (\log n)^{3+\epsilon}$, for fixed $\epsilon > 0$, in which case the bounds claimed are $O(\log n)$ time and $O(n)$ operations with high probability; the extension to smaller word lengths centers around a randomized version of Lemma 3 that is less wasteful, in terms of word length. One may note that Theorem 5 actually strengthens Theorem 2. The sequential version of Theorem 5 does not need the AKS network.

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. Given these numbers, T_D can be constructed in $O(\log n)$ time using $O(n)$ operations, as described in [11], and the Euler-tour technique [19] and optimal list ranking [7] can be used to collect the leaves of T_D in left-to-right order after the sorting at the internal nodes, which concludes the whole sorting.

We choose $k = \Theta(w/(\log n)^{2+\epsilon/2})$, which allows us to sort at the nodes of the trie in $O(\log n)$ time using the algorithm of Theorem 3. Since the "reduction factor" k/f is $\Omega((\log n)^{\epsilon/2})$, after a constant number of reduction steps we can also sort the concatenated signatures in $O(\log n)$ time using the algorithm of Theorem 3. □

Corollary 3 *If $w \geq (\log n)^{2+\epsilon}$ 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.*

Theorem 6 *For all integers $n \geq 4$, n multiple-precision integers occupying at most L machine words each and N machine words altogether can be sorted either in $O(L + \log n \log\log n)$ 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.*

PROOF Omitted. □

7 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 unrealistically large word sizes. 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 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*, pp. 1–9, 1983.

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

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

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

[5] 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*, pp. 538–546, 1993.

[6] 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, pp. 29–47, 1991.

[7] 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, pp. 128–142, 1988.

[8] 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.

[9] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. System Sci.*, 47, pp. 424–436, 1993.

[10] 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*, pp. 135–143, 1984.

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

[12] T. Hagerup and H. Shen. Improved nonconservative sequential and parallel integer sorting. *Inform. Proc. Lett.*, 36, pp. 57–63, 1990.

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

[14] D. Kirkpatrick and S. Reisch. Upper bounds for sorting integers on random access machines. *Theoret. Computer Science*, 28, pp. 263–276, 1984.

[15] D. E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley, Reading, MA, 1973.

[16] C. P. Kruskal. Searching, merging, and sorting in parallel computation. *IEEE Trans. Comput.*, 32, pp. 942–946, 1983.

[17] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann Publ., San Mateo, CA, 1992.

[18] W. J. Paul and J. Simon. Decision trees and random access machines. In *Proc. International Symp. on Logic and Algorithmic*, Zürich, pp. 331–340, 1980.

[19] R. E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM J. Comput.*, 14, pp. 862–874, 1985.