Expected Linear Time Sorting for Word Size $\Omega(\log^2 n \log\log n)$

Djamal Belazzougui[1⋆], Gerth Stølting Brodal[2], and Jesper Sindahl Nielsen[2]

[1] Helsinki Institute for Information Technology (hiit), Department of Computer Science, University of Helsinki
dbelaz@liafa.univ-paris-diderot.fr
[2] MADALGO[⋆⋆], Department of Computer Science, Aarhus University, Denmark
{gerth,jasn}@cs.au.dk

Abstract. Sorting n integers in the word-RAM model is a fundamental problem and a long-standing open problem is whether integer sorting is possible in linear time when the word size is $\omega(\log n)$. In this paper we give an algorithm for sorting integers in expected linear time when the word size is $\Omega(\log^2 n \log\log n)$. Previously expected linear time sorting was only possible for word size $\Omega(\log^{2+\varepsilon} n)$. Part of our construction is a new packed sorting algorithm that sorts n integers of w/b-bits packed in $\mathcal{O}(n/b)$ words, where b is the number of integers packed in a word of size w bits. The packed sorting algorithm runs in expected $\mathcal{O}(\frac{n}{b}(\log n + \log^2 b))$ time.

1 Introduction

Sorting is one of the most fundamental problems in computer science and has been studied widely in many different computational models. In the comparison based setting the worst case and average case complexity of sorting n elements is $\Theta(n \log n)$, and running time $\mathcal{O}(n \log n)$ is e.g. achieved by Mergesort and Heapsort [19]. The lower bound is proved using decision trees, see e.g. [4], and is also valid in the average case.

In the word-RAM model with word size $w = \Theta(\log n)$ we can sort n w-bit integers in $\mathcal{O}(n)$ time using radix sort. The exact bound for sorting n integers of w bits each using radix sort is $\Theta(n \frac{w}{\log n})$. A fundamental open problem is if we can still sort in linear time when the word size is $\omega(\log n)$ bits. The RAM dictionary of van Emde Boas [17] allows us to sort in $\mathcal{O}(n \log w)$ time. Unfortunately the space usage by the van Emde Boas structure cannot be bounded better than $\mathcal{O}(2^w)$. The space usage can be reduced to $\mathcal{O}(n)$ by using the Y-fast trie of Willard [18], but the time bound for sorting becomes expected. For polylogarithmic word sizes, i.e. $w = \log^{\mathcal{O}(1)} n$, this gives sorting in time $\mathcal{O}(n \log\log n)$. Kirkpatrick and Reisch gave an algorithm achieving $\mathcal{O}(n \log \frac{w}{\log n})$ [11], which also

⋆ Work done while visiting MADALGO.

⋆⋆ Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation (grant DNRF84).

gives $\mathcal{O}(n \log \log n)$ for $w = \log^{\mathcal{O}(1)} n$. Andersson et al. [3] showed how to sort in expected $\mathcal{O}(n)$ time for word size $w = \Omega(\log^{2+\varepsilon} n)$ for any $\varepsilon > 0$. The result is achieved by exploiting word parallelism on "signatures" of the input elements packed into words, such that a RAM instruction can perform several element operations in parallel in constant time. Han and Thorup [10] achieved running time $\mathcal{O}(n\sqrt{\log(w/\log n)})$, implying the best known bound of $\mathcal{O}(n\sqrt{\log \log n})$ for sorting integers that is independent of the word size. Thorup established that maintaining RAM priority queues and RAM sorting are equivalent problems by proving that if we can sort in time $\mathcal{O}(n \cdot f(n))$ then there is a priority queue using $\mathcal{O}(f(n))$ time per operation [15].

Our results. We consider for which word sizes we can sort n w-bit integers in the word-RAM model in expected linear time. We improve the previous best word size of $\Omega(\log^{2+\varepsilon} n)$ [3] to $\Omega(\log^2 n \log \log n)$. Word-level parallelism is used extensively and we rely on a new packed sorting algorithm (see Section 5) in intermediate steps. The principal idea for the packed sorting algorithm is an implementation of the randomized Shell-sort of Goodrich [7] using the parallelism in the RAM model. The bottleneck in our construction is $\mathcal{O}(\log \log n)$ levels of packed sorting of $\mathcal{O}(n)$ elements each of $\Theta(\log n)$ bits, where each sorting requires time $\mathcal{O}(n \frac{\log^2 n}{w})$. For $w = \Omega(\log^2 n \log \log n)$, the overall time becomes $\mathcal{O}(n)$.

This paper is structured as follows: Section 2 contains a high level description of the ideas and concepts used by our algorithm. In Section 3 we summarize the RAM operations adopted from [3] that are needed to implement the algorithm outlined in Section 2. In Section 4 we give the details of implementing the algorithm on a RAM and in Section 5 we present the packed sorting algorithm. Finally, in Section 6 we discuss how to adapt our algorithm to work with an arbitrary word size.

2 Algorithm

In this section we give a high level description of the algorithm. The input is n words $x_1, x_2, \ldots, x_n$, each containing a w-bit integer from $U = \{0, 1, \ldots, 2^w - 1\}$. We assume the elements are distinct. Otherwise we can ensure this by hashing the elements into buckets in expected $\mathcal{O}(n)$ time and only sorting a reduced input with one element from each bucket. The algorithm uses a Monte Carlo procedure, which sorts the input with high probability. While the output is not sorted, we repeatedly rerun the Monte Carlo algorithm, turning the main sorting algorithm into a Las Vegas algorithm.

The Monte Carlo algorithm is a recursive procedure using geometrically decreasing time in the recursion, ensuring $\mathcal{O}(n)$ time overall. We view the algorithm as building a Patricia trie over the input words by gradually refining the Patricia trie in the following sense: on the outermost recursion level characters are considered to be w bits long, on the next level $w/2$ bits, then $w/4$ bits and so on. The main idea is to avoid considering all the bits of an element to decide its rank. To avoid looking at every bit of the bit string e at every level of the recursion, we either consider the MSH(e) (Most Significant Half, i.e. the $\frac{|e|}{2}$ most significant bits of e) or LSH(e) (Least Significant Half) when moving one level down in the recursion (similar to the recursion in van Emde Boas trees).

The input to the ith recursion is a list $(id_1, e_1), (id_2, e_2), \ldots, (id_m, e_m)$ of length m, where $n \leq m \leq 2n-1$, id_j is a $\log n$ bit id and e_j is a $w/2^i$ bit element. At most n elements have equal id. The output is a list of ranks $\pi_1, \pi_2, \ldots, \pi_m$, where the j'th output is the rank of e_j among elements with id identical to id_j using $\log n$ bits. There are $m(\log n + \frac{w}{2^i})$ bits of input to the ith level of recursion and $m \log n$ bits are returned from the ith level. On the outermost recursion level we take the input $x_1, x_2, \ldots, x_n$ and produce the list $(1, x_1), (1, x_2), \ldots, (1, x_n)$, solve this problem, and use the ranks $\pi_1, \pi_2, \ldots, \pi_n$ returned to permute the input in sorted order in $\mathcal{O}(n)$ time.

To describe the recursion we need the following definitions.

Definition 1 ([6]). *The* Patricia trie *consists of all the branching nodes and leaves of the corresponding compacted trie as well as their connecting edges. All the edges in the Patricia trie are labeled only by the first character of the corresponding edge in the compacted trie.*

Definition 2. *The Patricia trie of $x_1, x_2, \ldots, x_n$ of* detail i*, denoted T^i, is the* Patricia trie *of $x_1, \ldots, x_n$ when considered over the alphabet $\Sigma^i = \{0,1\}^{w/2^i}$.*

The input to the ith recursion satisfies the following invariants, provided the algorithm has not made any errors so far:

i. The number of bits in an element is $|e| = \frac{w}{2^i}$.
ii. There is a bijection from id's to non leaf nodes in T^i.
iii. The pair (id, e) is in the input if and only if there is an edge from a node $v \in T^i$ corresponding to id to a child labeled by a string in which $e \in \Sigma^i$ is the first character.

That the maximum number of elements at any level in the recursion is at most $2n-1$ follows because a Patricia trie on n strings has at most $2n-1$ edges.

The recursion. The base case of the recursion is when $|e| = \mathcal{O}(\frac{w}{\log n})$ bits, i.e. we can pack $\Omega(\log n)$ elements into a single word, where we use the packed sorting algorithm from Section 5 to sort (id_j, e_j, j) pairs lexicographically by (id, e) in time $\mathcal{O}(\frac{n}{\log n}(\log n + (\log \log n)^2)) = \mathcal{O}(n)$. Then we generate the ranks π_j and return them in the correct order by packed sorting pairs (j, π_j) by j.

When preparing the input for a recursive call we need to halve the number of bits the elements use. To maintain the second invariant we need to find all the branching nodes of T^{i+1} to create a unique id for each of them. Finally for each edge going out of a branching node v in T^{i+1} we need to make the pair (id, e), where id is v's id and e is the first character (in Σ^{i+1}) on an edge below v. Compared to level i, level $i+1$ may have two kinds of branching nodes: *inherited nodes* and *new nodes*, as detailed below (Figure 1).

In Figure 1 we see T^i and T^{i+1} on 5 bit-strings. In T^i characters are 4 bits and in T^{i+1} they are 2 bits. Observe that node a is not going to be a branching

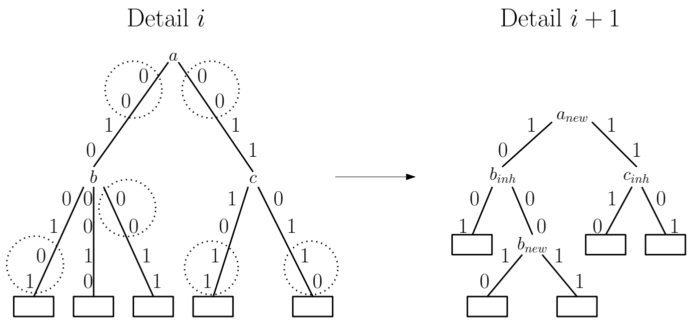


Fig. 1. Example of how nodes are introduced and how they disappear from detail i to $i+1$. The bits that are marked by a dotted circle are omitted in the recursion.

node when characters are 2 bits because "00" are the first bits on both edges below it. Thus the "00" bits below a should not appear in the next recursion – this is captured by Invariant iii. A similar situation happens at the node b, however since there are *two different* 2-bit strings below it, we get the inherited node b_{inh}. At the node c we see that the order among its edges is determined by the first two bits, thus the last two bits can be discarded. Note there are 7 elements in the ith recursion and 8 in the next – the number of elements may increase in each recursion, but the maximum amount is bounded by $2n-2$.

By invariant ii) every *id* corresponds to a node v in T^i. If we find all elements that share the same *id*, then we have all the outgoing edges of v. We refine an edge labeled e out of v to have the two characters MSH(e)LSH(e) both of $w/2^{i+1}$ bits. Some edges might then share their MSH. The node v will appear in level $i+1$ if and only if at least two outgoing edges do not share MSH – these are the *inherited nodes*. Thus we need only count the number of unique MSHs out of v to decide if v is also a node in level $i+1$. The edges out of v at level $i+1$ will be the unique MSH characters (in Σ^{i+1}) on the edges down from v at level i.

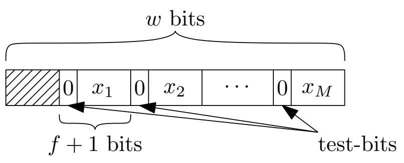
If at least two edges out of v share the same first character c (MSH), but not the second, then there is a branching node following c – these are the *new nodes*. We find all new nodes by detecting for each MSH character $c \in \Sigma^{i+1}$ going out of v if there are two or more edges with c as their first character. If so, we have a branching node following c and the labels of the edges are the LSHs. At this point everything for the recursion is prepared.

We receive for each id/node of T^{i+1} the ranks of all elements (labels on the outgoing edges) from the recursion. A *relative* rank for an element at level i is created by concatenating the rank of MSH(e) from level $i+1$ with the rank of LSH(e) from level $i+1$. All edges branching out of a new node needs to receive the rank of their MSH (first character). If the MSH was not used for the recursion, it means it did not distinguish any edges, and we can put an arbitrary value as the rank (we use 0). The same is true for the LSHs. Since each relative rank consists of $2\log n$ bits we can sort them fast using packed sorting (Section 5) and finally the actual ranks can be returned based on that.

3 Tools

This section is a summary of standard word-parallel algorithms used by our sorting algorithm; for an extensive treatment see [12]. In particular the prefix sum and word packing algorithms can be derived from [13]. For those familiar with "bit tricks" this section can be skipped.

We adopt the notation and techniques used in [3]. A w-bit word can be interpreted as a single integer in the range $0, \ldots, 2^w - 1$ or the interpretation can be parameterized by (M, f). A word under the (M, f) interpretation uses the rightmost $M(f + 1)$ bits as M fields using $f + 1$ bits each and the most significant bit in each field is called the *test bit* and is 0 by default.



We write $X = (x_1, x_2, \ldots, x_M)$ where x_i uses f bits, meaning the word X has the integer x_1 encoded in its leftmost field, x_2 in the next and so on. If $x_i \in \{0, 1\}$ for all i we may also interpret them as boolean values where 0 is false and 1 is true. This representation allows us to do "bit tricks".

Comparisons. Given a word $X = (x_1, x_2, \ldots, x_M)$ under the (M, f) interpretation, we wish to check $x_i > 0$ for $1 \leq i \leq M$, i.e. we want a word $Z = [X > 0] = (z_1, z_2, \ldots, z_M)$, in the (M, f) interpretation, such that $z_i = 1$ (true) if $x_i > 0$ and $z_i = 0$ (false) otherwise. Let $k_{M,f}$ be the word where the number k is encoded in each field where $0 \leq k < 2^f$. Create the word $0_{M,f}$ and set all test bits to 1. Evaluate $\neg(0_{M,f} - X)$, the ith test bit is 1 if and only if $x_i > 0$. By masking away everything but the test bit and shifting right by f bits we have the desired output. We can also implement more advanced comparisons, such as comparing $[X \leq Y]$ by setting all test bits to 1 in Y and 0 in X and subtracting the word X from Y. The test bits now equal the result of comparing $x_i \leq y_i$.

Hashing. We will use a family of hash functions that can hash n elements in some range $0, \ldots, m - 1$ with $m > n^c$ to $0, \ldots n^c - 1$. Furthermore a family of hash functions that are injective on a set with high probability when chosen uniformly at random, can be found in [5]. Hashing is roughly just multiplication by a random odd integer and keeping the most significant bits. The integer is at most f bits. If we just multiply this on a word in (M, f) interpretation one field might overflow to the next field, which is undesirable. To implement hashing on a word in (M, f) representation we first mask out all even fields, do the hashing, then do the same for odd fields. The details can be found in [3]. In [5] it is proved that if we choose a function h_a uniformly at random from the family $H_{k,\ell} = \{h_a \mid 0 < a < 2^k, \text{and } a \text{ is odd}\}$ where $h_a(x) = (ax \bmod 2^k) \operatorname{div} 2^{k-\ell}$ for $0 \leq x < 2^k$ then $\Pr[h_a(x) = h_a(y)] \leq \frac{1}{2^{\ell-1}}$ for distinct x, y from a set of size n. Thus choosing $\ell = c \log n + 1$ gives collision probability $\leq 1/n^c$. The probability that the function is not injective on n elements: $\Pr[\exists x, y : x \neq y \wedge h_a(x) = h_a(y)] \leq \frac{n^2}{n^c}$ (union bound on all pairs).

Prefix sum. Let $A = (a_1, \ldots, a_M)$ be the input with $M = b$, $f = w/b$ and $a_i \in \{0, 1\}$. In the output $B = (b_1, \ldots, b_M)$, $b_i = 0$ if $a_i = 0$ and $b_i = \sum_{j=1}^{i-1} a_j$ otherwise. We describe an $\mathcal{O}(\log b)$ time algorithm. The invariant is that in the jth iteration a_i has been added to its 2^j immediately right adjacent fields. Compute B_1, which is A shifted right by f bits and added to itself[3]: $B_1 = A + (A \downarrow f)$. Let $B_i = (B_{i-1} \downarrow 2^{i-1} f) + B_{i-1}$. This continues for $\log b$ steps. Then we keep all fields i from $B_{\log b}$ where $a_i = 1$, subtract 1 from all of these fields and return it.

Packing words. We are given a word $X = (x_1, \ldots, x_M)$ in $(M, f) = (b, w/b)$ representation. Some of the fields are zero fields, i.e. a field only containing bits set to 0. We want to produce a "packed word", such that reading from left to right there are no zero fields, followed only by zero fields. The fields that are nonzero in the input must be in the output and in the same order. This problem is solved by Andersson et al. [3, Lemma 6.4]

Expanding. Given a word with fields using b' bits we need to expand each field to using b bits i.e., given $X = (x_1, \ldots, x_k)$ where $|x_i| = b'$ we want $Y = (y_1, \ldots, y_k)$ such that $y_i = x_i$ but $|y_i| = b$. We assume there are enough zero fields in the input word such that the output is only one word. The general idea is to just do packing backwards. The idea is to write under each field the number of bits it needs to be shifted right, this requires at most $\mathcal{O}(\log b)$ bits per field. We now move items based on the binary representation. First we move those who have the highest bit set, then we continue with those that have the second highest bit set and so on. The proof that this works is the same as for the packing algorithm.

Creating index. We have a list of n elements of w/b bits each, packed in an array of words $X_1, X_2, \ldots, X_{n/b}$, where each word is in $(b, w/b)$ representation and $w/b \geq \lceil \log n \rceil$. Furthermore, the rightmost $\lceil \log n \rceil$ bits in every field are 0. The index of an element is the number of elements preceding it and we want to put the index in the rightmost bits of each field. First we will spend $\mathcal{O}(b)$ time to create the word $A = (1, 2, 3, \ldots, b)$ using the rightmost bits of the fields. We also create the word $B = (b, b, \ldots, b)$. Now we run through the input words, update $X_i = X_i + A$, then update $A = A + B$. The time is $\mathcal{O}(n/b + b)$, which in our case always is $\mathcal{O}(n/b)$, since we always have $b = \mathcal{O}(\log n \log \log n)$.

4 Algorithm – RAM details

In this section we describe how to execute each step of the algorithm outlined in Section 2. We first we describe how to construct T^{i+1} from T^i, i.e. advance one level in the recursion. Then we describe how to use the output of the recursion

[3] We use $\uparrow$ and $\downarrow$ as the shift operations where $x \uparrow y$ is $x \cdot 2^y$ and $x \downarrow y$ is $\lfloor x \text{ div } 2^y \rfloor$.

for T^{i+1} to get the ranks of the input elements for level i. Finally the analysis of the algorithm is given.

The input to the ith recursion is a list of pairs: (id, e) using $\log n + \frac{w}{2^i}$ bits each and satisfying the invariants stated in Section 2. The list is packed tightly in words, i.e. if we have m input elements they occupy $\mathcal{O}(\frac{m \cdot (\log n + w/2^i)}{w})$ words. The returned ranks are also packed in words, i.e. they occupy $\mathcal{O}(\frac{m \cdot \log n}{w})$ words. The main challenge of this section is to be able to compute the necessary operations, even when the input elements and output ranks are packed in words. For convenience and simplicity we assume tuples are not split between words.

Finding branching nodes. We need to find the branching nodes (*inherited* and *new*) of T^{i+1} given T^i. For each character e_j in the input list (i.e. T^i) we create the tuple (id_j, H_j, j) where id_j corresponds to the node e_j branches out of, $H_j = h(\text{MSH}(e_j))$ is the hash function applied to the MSH of e_j, and j is the index of e_j in the input list. The list L consists of all these tuples and L is sorted. We assume the hash function is injective on the set of input MSHs, which it is with high probability if $|H_j| \geq 4 \log n$ (see the analysis below). If the hash function is not injective, this step may result in an error which we will realize at the end of the algorithm, which was discussed in Section 2. The following is largely about manipulating the order of the elements in L, such that we can create the recursive sub problem, i.e. T^{i+1}.

To find *Inherited nodes* we find all the edges out of nodes that are in both T^i and T^{i+1} and pair them with unique identifiers for their corresponding nodes in T^{i+1}. Consider a fixed node a which is a branching node in T^i – this corresponds to an id in L. There is a node a_{inh} in T^{i+1} if a and its edges satisfy the following condition: When considering the labels of the edges from a to its children over the alphabet Σ^{i+1} instead of Σ^i, there are at least two edges from a to its children that do not share their first character. When working with the list L the node a and its edges correspond to the tuples where the id is the id that corresponds to a. This means we need to compute for each id in L whether there are at least 2 unique MSHs, and if so we need to extract precisely all the unique MSHs for that id.

The list L is sorted by (id_j, H_j, j), which means all edges out of a particular node are adjacent in L, and all edges that share their MSH are adjacent in L (with high probability), because they have the same hash value which is distinct from the hash value of all other MSHs (with high probability). We select the MSHs corresponding to the first occurrence of each unique hash value with a particular id for the recursion (given that it is needed). To decide if a tuple contains a first unique hash value, we need only consider the previous tuple: did it have a different hash value from the current, or did it have a different id? To decide if $\text{MSH}(e_j)$ should be extracted from the corresponding tuple we also need to compute whether there are at least two unique hash values with id id_j. This tells us we need to compute two things for every tuple (id_j, H_j, j) in L:

1. Is j the first index such that $(id_{j-1} = id_j \wedge H_{j-1} \neq H_j) \vee id_{j-1} \neq id_j$?
2. Is there an i such that $id_i = id_j$ and $H_i \neq H_j$?

To accomplish the first task we do parallel comparison of id_j and H_j with id_{j-1} and H_{j-1} on L and L shifted left by one tuple length (using the word-level parallel comparisons described in Section 3). The second task is tedious but conceptually simple to test: count the number of unique hash values for each id, and test for each id if there are at least two unique hash values.

The details of accomplishing the two tasks are as follows (keep in mind that elements of the lists are bit-strings). Let B be a list of length $|L|$ and consider each element in B as being the same length as a tuple in L. Encode 1 in element j of B if and only if $id_{j-1} \neq id_j$ in L. Next we create a list C with the same element size as B. There will be a 1 in element j of C if and only if $H_j \neq H_{j-1} \wedge id_j = id_{j-1}$ (this is what we needed to compute for task 1). The second task is now to count how many 1s there are in C between two ones in B. Let CC be the prefix sum on C (described in Section 3) and keep only the values where there is a corresponding 1 in B, all other elements become 0 (simple masking). Now we need to compute the difference between each non-zero value and the next non-zero value in CC – but these are varying lengths apart, how do we subtract them? The solution is to pack the list CC (see Section 3) such that the values become adjacent. Now we compute the difference, and by maintaining some information from the packing we can unpack the differences to the same positions that the original values had. Now we can finally test for the first tuple in each id if there are at least two different hash values with that id. That is, we now have a list D with a 1 in position j if j is the first position of an id in L and there are at least two unique MSHs with that id. In addition to completing the two tasks we can also compute the unique identifiers for the inherited nodes in T^{i+1} by performing a prefix sum on D.

Finding the *new nodes* is simpler than finding the *inherited nodes*. The only case where an LSH should be extracted is when two or more characters out of a node share MSH, in which case all the LSHs with that MSH define the outgoing edges of a *new node*. Observe that if two characters share MSH then their LSHs must differ, due to the assumption of distinct elements propagating through the recursion. To find the relevant LSHs we consider the sorted list L. Each *new node* is identified by a pair (id, MSH) where $(id_j, h(\text{MSH}(e_j)), \cdot)$ appears at least twice in L, i.e. two or more tuples with the same id and hash of MSH. For each *new node* we find the leftmost such tuple j in L.

Technically we scan through L and evaluate $(H_{j-1} \neq H_j \vee id_{j-1} \neq id_j) \wedge (H_{j+1} = H_j \wedge id_{j+1} = id_j)$. If this evaluates to true then j is a *new node* in T^{i+1}. Using a prefix sum we create and assign all ids for *new nodes* and their edges. In order to test if LSH_j should be in the recursion we evaluate $(H_{j-1} = H_j \wedge id_{j-1} = id_j) \vee (H_{j+1} = H_j \wedge id_{j+1} = id_j)$. This evaluates to true only if the LSH should be extracted for the recursion because we assume distinct elements.

Using results from the recursion. We created the input to the recursion by first extracting all MSHs, packing them and afterwards extracting all LSHs and then packing them. Finally concatenate the two packed arrays. Now we simply have to reverse this process, first for the MSHs, then the LSHs. Technically after the recursive call the array of tuples $(j, rank_{\mathrm{MSH}_j}, rank_{\mathrm{LSH}_j}, H_j, id_j, rank_{new})$ is filled out. Some of the fields are just additional fields to the array L. The three ranks use $\log n$ bits each and are initialized to 0. First $rank_{\mathrm{MSH}_j}$ is filled out and afterwards $rank_{\mathrm{LSH}_j}$. The same procedure is used for both.

For retrieving $rank_{\mathrm{MSH}_i}$, we know how many MSHs were extracted for the recursion, so we separate the ranks of MSHs and LSHs and now only consider MSHs ranks. We first expand the MSH ranks as described in Section 3 such that each rank uses the same number of bits as an entire tuple. Recall that the MSHs were packed and we now need to unpack them. If we saved information on how we packed elements, we can also unpack them. The information we need to retain is how many elements each word contributed and for each element in a word its initial position in that word. Note that for each unique H_j we only used one MSH for the recursion, thus we need to propagate its rank to all other elements with the same hash and id. Fortunately the hash values are adjacent, and by noting where the hash values change we can do an operation similar to a prefix sum to copy the ranks appropriately.

Returning. As this point the only field not filled out is $rank_{new}$. To fill it out we sort the list by the concatenation of $rank_{\mathrm{MSH}_j}$ and $rank_{\mathrm{LSH}_j}$. In this sorted list we put the current position of the elements in $rank_{new}$ (see Section 3 on creating index). The integer in $rank_{new}$ is currently not the correct rank, but by subtracting the first $rank_{new}$ in an id from the other $rank_{new}$s with that id we get the correct rank. Then we sort by j, mask away everything except $rank_{new}$, pack the array and return. We are guaranteed the ranks from the recursion use $\log n$ bits each, which means the concatenation uses $2\log n$ bits so we can sort the array efficiently.

Analysis. We argue that the algorithm is correct and runs in linear time.

Lemma 1. *Let n be the number of integers we need to sort then the maximum number of elements in any level of the recursion is $2n-1$.*

Proof. This follows immediately from the invariants. □

Theorem 1. *The main algorithm runs in $\mathcal{O}(n)$ time.*

Proof. At level i of the recursion $|e| = \frac{w}{2^i}$. After $\log\log n$ levels we switch to the base case where there are $b = 2^{\log\log n} = \log n$ elements per word. The time used in the base case is $\mathcal{O}(\frac{n}{b}(\log^2 b + \log n)) = \mathcal{O}(\frac{n}{\log n}((\log\log n)^2 + \log n)) = \mathcal{O}(n)$.

At level i of the recursion we have $b = 2^i$ elements per word and the time to work with each of the $\mathcal{O}(\frac{n}{b})$ words using the methods of Section 3 is $\mathcal{O}(\log b)$. The packed sorting at each level sorts elements with $\mathcal{O}(\log n)$ bits, i.e. $\mathcal{O}\left(\frac{w}{\log n}\right)$ elements per word in time $\mathcal{O}\left(\frac{n}{w/\log n}\left(\log^2 \frac{w}{\log n} + \log n\right)\right)$. Plugging in our assumption $w = \Omega(\log^2 n \log\log n)$, we get time $\mathcal{O}\left(\frac{n}{\log\log n}\right)$. For all levels the total time becomes $\sum_{i=0}^{\log\log n}\left(\frac{n}{2^i} i + \frac{n}{\log\log n}\right) = \mathcal{O}(n)$. □

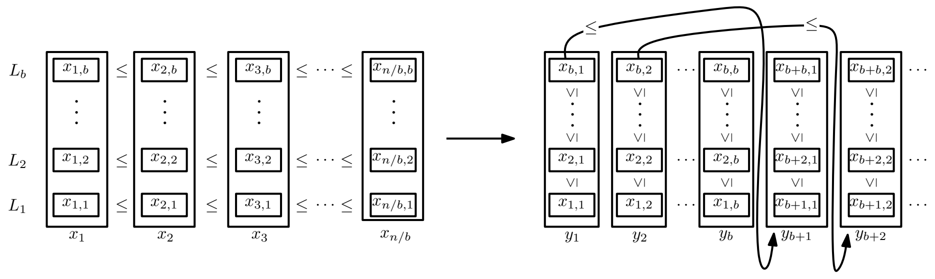


Fig. 2. Transposing and concatenating blocks.

The probability of doing more than one iteration of the algorithm is the probability that there is a level in the recursion where the randomly chosen hash function was not injective. The hash family can be designed such that the probability of a hash function not being injective when chosen uniformly at random is less than $1/n^2$ [5]. We need to choose $\log\log n$ such functions. The probability that at least one of the functions is not injective is $\mathcal{O}(\log\log n/n^2) < \mathcal{O}(1/n)$. In conclusion the sorting step works with high probability, thus we expect to repeat it $\mathcal{O}(1)$ times.

5 Packed sorting

We are given n elements of $\frac{w}{b}$ bits packed into $\frac{n}{b}$ words using $(M, f) = (b, w/b)$ representation that we need to sort. Albers and Hagerup [2] describe how to perform a deterministic packed sorting in time $\mathcal{O}(\frac{n}{b}\log n \cdot \log b)$. We describe a simple randomized word-level parallel sorting algorithm running in time $\mathcal{O}(\frac{n}{b}(\log n + \log^2 b))$. Packed sorting proceeds in four steps described in the following sections. The idea is to implement b sorting networks in parallel using word-level parallelism. In sorting networks one operation is available: compare the elements at positions i and j then swap i and j based on the outcome of the comparison. Denote the ℓth element of word i at any point by $x_{i,\ell}$. First we use the ℓth sorting network to get a sorted list L_ℓ: $x_{1,\ell} \leq x_{2,\ell} \leq \cdots \leq x_{n/b,\ell}$ for $1 \leq \ell \leq b$. Each L_ℓ then occupies field ℓ of every word. Next we reorder the elements such that each of the b sorted lists uses n/b^2 consecutive words, i.e. $x_{i,j} \leq x_{i,j+1}$ and $x_{i,w/b} \leq x_{i+1,1}$, where $n/b^2 \cdot k < i \leq n/b^2 \cdot (k+1)$ and $0 \leq k \leq b-1$ (See Figure 2). From that point we can merge the lists using the RAM implementation of bitonic merging (see below). The idea of using sorting networks or *oblivious* sorting algorithms is not new (see e.g. [9]), but since we need to sort in sublinear time (in the number of elements) we use a slightly different approach.

Data-oblivious sorting. A famous result is the AKS deterministic sorting network which uses $\mathcal{O}(n \log n)$ comparisons [1]. Other deterministic $\mathcal{O}(n \log n)$ sorting networks were presented in [2, 8]. However, in our application randomized sorting suffices so we use the simpler randomized Shell-sort by Goodrich [7]. An alternative randomized sorting-network construction was given by Leighton and Plaxton [14].

Randomized Shell-sort sorts any permutation with probability at least $1 - 1/N^c$ ($N = n/b$ is the input size), for any $c \geq 1$. We choose $c = 2$. The probability that b arbitrary lists are sorted is then at least $1 - b/N^c \geq 1 - N^{c-1}$. We check that the sorting was correct for all the lists in time $\mathcal{O}(\frac{n}{b})$. If not, we redo the oblivious sorting algorithm. Overall the expected running time is $\mathcal{O}(\frac{n}{b} \log \frac{n}{b})$.

The Randomized Shell-sort algorithm works on any adversarial chosen permutation that does not know the random choices of the algorithm. The algorithm uses randomization to generate a sequence of $\Theta(n \log n)$ comparisons (a sorting network) and then applies the sequence of comparisons to the input array. We start the algorithm of Goodrich [7] to get the sorting network. We run it with $N = n/b$ as the input size. When the network compares i and j, we compare words i and j field-wise. That is, the first element of the two words are compared, the second element of the words are compared and so on. Using the result we can implement the swap that follows. After this step we have $x_{1,\ell} \leq x_{2,\ell} \leq \cdots \leq x_{n/b,\ell}$ for all $1 \leq \ell \leq b$.

The property of Goodrich' Shellsort that makes it possible to apply it in parallel is its data obliviousness. In fact any sufficiently fast data oblivious sorting algorithm would work.

Verification step. The verification step proceeds in the following way: we have n/b words and we need to verify that the words are sorted field-wise. That is, to check that $x_{i,\ell} \leq x_{i+1,\ell}$ for all i, ℓ. One packed comparison will be applied on each pair of consecutive words to verify this. If the verification fails, then we redo the oblivious sorting algorithm.

Rearranging the sequences. The rearrangement in Figure 2 corresponds to looking at b words as a $b \times b$ matrix (b words with b elements in each) and then transposing this matrix. Thorup [16, Lemma 9] solved this problem in $\mathcal{O}(b \log b)$ time. We transpose every block of b consecutive words. The transposition takes overall time $\mathcal{O}(\frac{n}{b} \log b)$. Finally, we collect in correct order all the words of each run. This takes time $\mathcal{O}(\frac{n}{b})$. Building the ith run for $1 \leq i \leq b$ consists of putting together the ith words of the blocks in the block order. This can be done in a linear scan in $\mathcal{O}(n/b)$ time.

Bitonic merging. The last phase is the bitonic merging. We merge pairs of runs of $\frac{n}{b^2}$ words into runs of $\frac{2n}{b^2}$ words, then runs of $\frac{2n}{b^2}$ words into runs of size $\frac{4n}{b^2}$ and so on, until we get to a single run of n/b words. We need to do $\log b$ rounds, each round taking time $\mathcal{O}(\frac{n}{b} \log b)$ making for a total time of $\mathcal{O}(\frac{n}{b} \log^2 b)$ [2].

6 General sorting

In this section we tune the algorithm slightly and state the running time of the tuned algorithm in terms of the word size w. We see that for some word sizes we can beat the $\mathcal{O}(n\sqrt{\log \log n})$ bound. We use the splitting technique of [10, Theorem 7] that given n integers can partition them into sets $X_1, X_2, \ldots X_k$ of at most $\mathcal{O}(\sqrt{n})$ elements each, such that all elements in X_i are less than all elements in X_{i+1} in $\mathcal{O}(n)$ time. Using this we can sort in $\mathcal{O}(n \log \frac{\log n}{\sqrt{w/\log w}})$ time.

The algorithm repeatedly splits the set S of inital size n_0 into smaller subsets of size $n_j = \sqrt{n_{j-1}}$ until we get $\log n_j \le \sqrt{w/\log w}$ where it stops and sorts each subset in linear time using our sorting algorithm. The splitting is performed $\log((\log n)/(\sqrt{w/\log w})) = \frac{1}{2}\log\frac{\log^2 n \log w}{w} = \mathcal{O}(\log\frac{\log^2 n \log\log n}{w})$ times. An interesting example is to sort in time $\mathcal{O}(n \log\log\log n)$ for $w = \frac{\log^2 n}{(\log\log n)^c}$ for any constant c. When $w = \frac{\log^2 n}{2^{\Omega(\sqrt{\log\log n})}}$, the sorting time is $\Omega(n\sqrt{\log\log n})$.

References

1. M. Ajtai, J. Komlós, and E. Szemerédi. An $\mathcal{O}(n \log n)$ sorting network. In *STOC*, pages 1–9, 1983.
2. S. Albers and T. Hagerup. Improved parallel integer sorting without concurrent writing. *Inf. Comput.*, 136(1):25–51, 1997.
3. A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in linear time? *Journal of Computer and System Sciences*, 57:74–93, 1998.
4. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press and McGraw Hill, 3rd edition, 2009.
5. M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. A reliable randomized algorithm for the closest-pair problem. *J. Algorithms*, 25(1):19–51, 1997.
6. P. Ferragina and R. Grossi. The string B-tree: A new data structure for string search in external memory and its applications. *J. ACM*, 46(2):236–280, 1999.
7. M. T. Goodrich. Randomized shellsort: A simple data-oblivious sorting algorithm. *J. ACM*, 58(6):27, 2011.
8. M. T. Goodrich. Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in $\mathcal{O}(n \log n)$ time. *CoRR*, abs/1403.2777, 2014.
9. T. Hagerup. Sorting and searching on the word RAM. In *STACS*, pages 366–398, 1998.
10. Y. Han and M. Thorup. Integer sorting in $\mathcal{O}(n\sqrt{\log\log n})$ expected time and linear space. In *FOCS*, pages 135–144, 2002.
11. D. Kirkpatrick and S. Reisch. Upper bounds for sorting integers on random access machines. *Theoretical Computer Science*, 28(3):263–276, 1983.
12. D. E. Knuth. *The Art of Computer Programming*, volume 4A: Combinatorial Algorithms. Addison-Wesley Professional, 2011.
13. F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, chapter 3.4.3 Packing, Spreading, and Monotone Routing Problems. Morgan Kaufmann Publishers, Inc., 1991.
14. T. Leighton and C. G. Plaxton. Hypercubic sorting networks. *SIAM Journal on Computing*, 27(1):1–47, 1998.
15. M. Thorup. On RAM priority queues. *SIAM J. Comput.*, 30(1):86–109, 2000.
16. M. Thorup. Randomized sorting in $\mathcal{O}(n \log\log n)$ time and linear space using addition, shift, and bit-wise boolean operations. *J. Alg.*, 42(2):205–230, 2002.
17. P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *FOCS*, pages 75–84, 1975.
18. D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(n)$. *Inf. Process. Lett.*, 17(2):81–84, 1983.
19. J. W. J. Williams. Algorithm 232: Heapsort. *CACM*, 7(6):347–348, 1964.