A Linear Time Algorithm for Ordered Partition

Yijie Han

School of Computing and Engineering
University of Missouri at Kansas City
Kansas City, Missouri 64110
hanyij@umkc.edu

Abstract. We present a deterministic linear time and space algorithm for ordered partition of a set T of n integers into $n^{1/2}$ sets $T_0 \leq T_1 \leq \cdots \leq T_{n^{1/2}-1}$, where $|T_i| = \theta(n^{1/2})$ and $T_i \leq T_{i+1}$ means that $\max T_i \leq \min T_{i+1}$.
Keywords: Algorithms, sorting, integer sorting, linear time algorithm, ordered partition, hashing, perfect hash functions.

1 Introduction

For a set T of n input integers we seek to partition them into $n^{1/2}$ sets $T_0, T_1, ..., T_{n^{1/2}} - 1$ such that $|T_i| = \theta(n^{1/2})$, $T_i \leq T_{i+1}$, $0 \leq i < n^{1/2} - 1$. Where $T_i \leq T_{i+1}$ means $\max T_i \leq \min T_{i+1}$. We call this ordered partition. We show that we can do this in deterministic optimal time, i.e. in $O(n)$ time.

This result, when applied iteratively for $O(\log\log n)$ iterations, partitions the n integers into $O(n^{3/4})$ sets because every set is further partitioned into $O(n^{1/4})$ sets, into $O(n^{7/8})$ sets, and so on, eventually partitions n integers into n ordered sets, i.e., having them sorted. The time for these iterations is $O(n \log\log n)$ and the space complexity is linear, i.e. $O(n)$. This complexity result for integer sorting was known [6] and is the current best result for deterministic integer sorting. However, ordered partition itself is an interesting topic for study and the result for ordered partition can be applicable in the design of algorithms for other problems. As an example in [7] our ordered partition algorithm is extended for obtaining an improved randomized integer sorting algorithm.

The problem of ordered partition was noticed in [11] and the linear time complexity was conceived there. However, the deterministic linear time ordered partition algorithm presented here has a nice structure and the mechanism for the design of our algorithm is particularly worth noting. In particular the conversion of the randomized signature sorting to the deterministic version shown in Section 3.2 and the mechanism shown there were not explained clearly in [11]. Besides, our ordered partition algorithm is an optimal algorithm and it has been extended to obtain a better randomized algorithm [7] for integer sorting. Therefore we present them here for the sake of encouraging future research toward an optimal algorithm for integer sorting.

2 Overview

Integers are sorted into their destination partition by moving them toward their destination partition. One way of moving integers is to use the ranks computed for them to move them, say a rank r is obtained for an integer i then move i to position r. We will call such moving as moving by indexing. Note that because our algorithm is a linear time algorithm such move by indexing can happen only a constant number of times for each integer. Since a constant number of moving by indexing is not sufficient through our algorithm we need to use the packed moving by indexing, i.e. we pack a integers into a word with each integer having the rank of $\log n/(2a)$ bits (and thus the total number of bits for all ranks of packed integers in a word is $\log n/2$) and move these integers to the destination specified by these $\log n/2$ bits in one step.

Here we packed a integers into a word in order to move them by indexing. In order to have a integers packed in a word we need to reduce the input integers to smaller valued integers. This can be done in the randomized setting by using the signature sorting [1] which basically says:

Lemma 1 [1]: In the randomized setting, sorting of n $p\log n$ bits integers can be done with 2 passes of sorting n $p^{1/2}\log n$ bits integers.

However, because our algorithm is a deterministic algorithm we converted Lemma 1 to Lemma 2.

Lemma 2: Ordered partition of n $p\log n$ bits integers into $n^{1/2}$ partitions can be done with 2 passes of ordered partition of n $p^{1/2}\log n$ bits integers into $n^{1/2}$ partitions.

Lemma 2 is basically proved in Section 3.2 as Lemma $2'$ and it is a deterministic algorithm. This is one of our main contributions.

By Lemma 2 we can assume that the integers we are dealing with has $p\log n$ bits while each word has $p^3\log n$ bits and thus we can pack more than p integers into a word.

We do ordered partition of n integers into $n^{1/2}$ partitions by a constant number of passes of ordered partition of n integers into $n^{1/8}$ partitions, i.e. ordered partition of n integers into $n^{1/8}$ partitions, into $n^{1/8+(1/8)(7/8)}$ partitions, ..., into $n^{1/2}$ partitions.

We do ordered partition of n integers in set T into $n^{1/8}$ partitions via ordered partition of n integers in T by $n^{1/8}$ integers $s_0 \le s_1 \le \cdots \le s_{n^{1/8}-1}$ in S, i.e. partition T into $|S|+1$ sets $T_0, T_1, ..., T_{|S|}$ such that $T_0 \le s_0 \le T_1 \le \cdots \le s_{|S|-1} \le T_{|S|}$. We call this as ordered partition of T by S.

The ordered partition is done by *arbitrary* partition T into $n^{1/2}$ sets $T(i)$'s, $0 \le i < n^{1/2}$, with $|T(i)| = n^{1/2}$ and pick arbitrary $n^{1/16}$ integers $s_0, s_1, ..., s_{n^{1/16}-1}$ in T to form set S. Then use step i to do ordered partition of $T(i)$ by S, $i = 0, 1, ..., n^{1/2}-1$. After step $i-1$ we have ordered partitioned $\cup_{j=0}^{i-2} T(j)$ by S into $T_0 \le s_0 \le T_1 \le s_1 \le \cdots \le s_{n^{1/16}-1} \le T_{n^{1/16}}$ and we maintain that $|T_j| \le 2n^{15/16}$ for all j. If after step i $2n^{15/16} < |T_j| \le 2n^{15/16} + n^{1/2}$ then we use selection [4] in $O(|T_j|)$ time to select the median m of T_j and (ordered) partition T_j into two sets. We add m to S. The next time a partition has to be ordered partitioned into two sets is after adding additional $n^{15/16}+n^{1/2}$ integers (from $n^{15/16}$ integers to $2n^{15/16}+n^{1/2}$ integers). Thus the overall time of ordered partitioning of one set (partition) to two sets (partitions) is linear. This idea is shown in [2]. Thus after finishing we have partitioned T into $O(n^{1/16})$ sets with each set of size $\le 2n^{15/16}$. We can then do selections for these sets to have them become $n^{1/16}$ ordered partitioned sets with each set of size $n^{15/16}$. The time is linear.

Thus we now left with the problem of ordered partition of T with $|T| = n$ by S with $|S| = n^{1/8}$. This is a complicated part of our algorithm. We achieve this using the techniques of deterministic perfect hashing, packed moving by indexing or sorting, progressive enlarge the size of hashed integers for packed moving and the determination of progress when hashed value matches, etc.. The basic idea is outline in the next two paragraphs.

Let $s_0 \le s_1 \le \cdots \le s_{n^{1/8}-1}$ be the integers in S. As will be shown in Section 4 that we can compare all integers in T with $s_i, s_{2i}, s_{3i}, ..., s_{\lfloor n^{1/8}/i \rfloor i}$ in $O(n\log(n^{1/8}/i)/\log n)$ time because integers can be hashed to $O(\log(n^{1/8}/i))$

bits (this is a perfect hash for $s_i, s_{2i}, s_{3i}, ..., s_{\lfloor n^{1/8}/i \rfloor i}$ but not perfect for the integers in T, but the hash value for every integer has $O(\log(n^{1/8}/i))$ bits) and we can pack $O(\log n / \log(n^{1/8}/i)))$ integers into one word (note that we can pack more integers into one word but the number of hashed bits will exceed $\log n/2$ which cannot be handled by our algorithm). Because of this packing we left with $O(n \log(n^{1/8}/i)/\log n)$ words and we can sort them using bucket sorting (by treating the hashed bits together as one integer in each word) in $O(n \log(n^{1/8}/i)/\log n)$ time. This sorting will let us compare all integers in T with $s_i, s_{2i}, s_{3i}, ..., s_{\lfloor n^{1/8}/i \rfloor i}$. As noted in Section 4 if an integer t in T is equal to s_{ai} for some a (we call match) then we made progress (we say advanced). Otherwise t is not equal to any s_{ai} (we call unmatch) and then we will compare these unmatched t's with $s_j, s_{2j}, s_{3j}, ..., s_{\lfloor n^{1/8}/j \rfloor j}$, where $n^{1/8}/j = (n^{1/8}/i)^2$ (i.e. we double the number of bits for hashed value for every integer). We keep doing this when unmatch happens till i (or j) becomes 1 where we have compared t with all integers in S and none of them is equal to t.

If for every integer t in T we did the last paragraph then if t does not match any integer in S we will lose our work because we will not know which partition t will fall into. To prevent this to happen when we compare t to integers in S we really do is to compare the most significant $\log n$ bits of t to the most significant $\log n$ bits of integers in S. And thus if the most significant $\log n$ bits of t does not match the most significant $\log n$ bits of any integer in S we can then throw away all bits of t except the most significant $\log n$ bits. We can then bucket sort t (because there are only $\log n$ bits for t left) into its destination partition. The complication of this scheme such as when the most significant $\log n$ bits match is handled in Section 4 where we used the concept of current segment for every integer. Initially the most significant $\log n$ bits of an integer is the current segment of the integer. When the current segment matches we then throw away the current segment and use the next $\log n$ bits as the current segment. The details is shown in Section 4. .

3 Preparation

3.1 Perfect Hash Function

We will use a perfect hash function H that can hash a set S of integers in $\{0, 1, ..., m-1\}$ to integers in $\{0, 1, ..., |S|^2 - 1\}$. A perfect hash function is a hash function that has no collisions for the input set. Such a hash function for S can be found in $O(|S|^2 \log m)$ time [13]. In [8] we improved this and made the time independent of the number of bits in integers ($\log m$):

Lemma 3 [8]: A perfect hash function that hashes a set S of integers in $\{0, 1, ..., m-1\}$ to integers in $\{0, 1, ..., |S|^2 - 1\}$ can be found in $O(|S|^4 \log |S|)$ time. Thereafter every batch of $|S|$ integers can be hashed to $\{0, 1, ..., |S|^2 - 1\}$ in $O(|S|)$ time and space. □

We note that the hash function used in this paper was first devised in [5] where it is showed that such a hash function can be found with a randomized algorithm in constant time. Raman [13] showed how to use derandomization to obtain a deterministic version of the hash function in $O(n^2 \log m)$ time for a set of n integers in $\{0, 1, ..., m-1\}$.

The hash function given in [5, 13] for hashing integers in $\{0, 1, ..., 2^k - 1\}$ to $\{0, 1, ..., 2^l - 1\}$ is

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

with different values of a different hash functions are defined. Raman [13] showed how to obtain the value of a such that the hash function becomes a deterministic perfect hash function. Note that the computation of hash values for multiple integers packed in a word can be done in constant time.

3.2 Converting Ordered Partition with Integers of $p \log n$ Bits to That with Integers of $p^{1/3} \log n$ Bits

Besides ordered partition, we now introduce the problem of ordered partition of set T of n integers by a set S of $|S| < n$ integers. Let $s_0 \le s_1 \le \cdots \le s_{|S|-1}$ be the $|S|$ integers in S that are already sorted. Ordered partition T by S is to ordered partition T into $|S|+1$ sets $T_0 \le T_1 \le \cdots \le T_{|S|}$ such that $T_0 \le s_0 \le T_1 \le \cdots \le s_{|S|-1} \le T_{|S|}$, where $T_i \le s_i$ means $\max T_i \le s_i$ and $s_i \le T_{i+1}$ means $s_i \le \min T_{i+1}$.

In our application in the following sections, $|S| = n^{1/8}$. Thus sorting set S takes at most $O(|S| \log |S|) = O(n)$ time. That is the reason we can consider the sorting of S as free as it will not dominate the time complexity of our algorithm.

We use a perfect hash function H for set S. Although there are many perfect hash functions given in many papers, we use the hash function in [5, 13] because it has the property of hashing multiple integers packed in one word in constant time. Notice that H hashes S into $\{0, 1, ..., |S|^2 - 1\}$ instead of $\{0, 1, ..., c|S| - 1\}$ for any constant c.

We shall use H to hash integers in T. Although H is perfect for S it is not perfect for T as T has n integers. We cannot afford to compute a perfect hash function for T as it requires nonlinear time. The property we will use is that for any two integers in S if their hash values are different then these two integers are different. For any integer $t \in T$ if the hash value of t is different than the hash value of an integer s in S then $t \ne s$.

We will count the bits in an integer from the low order bits to the high order bits starting at the least significant bit as the 0-th bit.

For sorting integers in $\{0, 1, ..., m-1\}$ we assume that the word size (the number of bits in a word) is $\log m + \log n$. $\log n$ bits are needed here for indexing into n input numbers. When $m = n^k$ for a constant k n integers can be sorted in linear time using radix sort. Let $p = \log m / \log n$ and therefore we have $p \log n$ bits for each integer.

As will be shown, we can do ordered partition of T by S, where $|T| = n$ and $|S| \le n$, (here integers in $T \cup S$ are taken from $\{0, 1, ..., m-1\}$, i.e. each integer has $p \log n$ bits) in constant number of passes with each pass being an ordered partition of set T'_i by S'_i, $i = 1, 2, 3, ..., c$, where T'_i has n integers each having $p^{1/3} \log n$ bits and S'_i has $|S|$ integers each having $p^{1/3} \log n$ bits, c is a constant.

This is done as follows. We view each integer of $p \log n$ bits in T (S) as composed of $p^{1/3}/2$ segments with each segment containing consecutive $2p^{2/3} \log n$ bits. We use a perfect hash function H which hashes each segment of an integer in S into $2 \log |S|$ bits and thus each integer in S is hashed to $p^{1/3} \log n$ bits. Note that the hash function provided in [5, 13] can hash all segments contained in a word in constant time (It requires [5, 13] that for each segment g of $2p^{2/3} \log n$ bits another $2p^{2/3} \log n$ bits g_1 has to be reserved and two segments of bits $g_1 g$ is used for the hashing of segment g. This can be achieved by separating even indexed segments into a word and odd indexed segments into another word).

We then use H to hash (segments of) integers in T and thus each integer in T is also hashed to $p^{1/3} \log n$ bits. Note here H is not perfect with respect to the (segments of) integers in T and thus different (segments) of integers in T may be hashed to the same hash value.

We use T'_1 (S'_1) to denote this hashed set, i.e. each integer in T'_1 (S'_1) has only $p^{1/3} \log n$ bits. Now assume that integers in S'_1 are sorted. If integer s (t) in S (T) is hashed to integer s' (t') in S'_1 (T'_1) then we use $H(s) = s'$ ($H(t) = t'$) to denote this. If s is in S then we also have that $s = H^{-1}(s')$. Note that if t is an integer in T and s is an integer in S, then if $H(t) = H(s)$ t may not be equal to s as H is not perfect for $T \cup S$. But if $H(t) \ne H(s)$ then $t \ne s$ and this is the property we will make use of. We will also use $H(S)$ ($H(T)$) to denote the set of integers hashed from S (T).

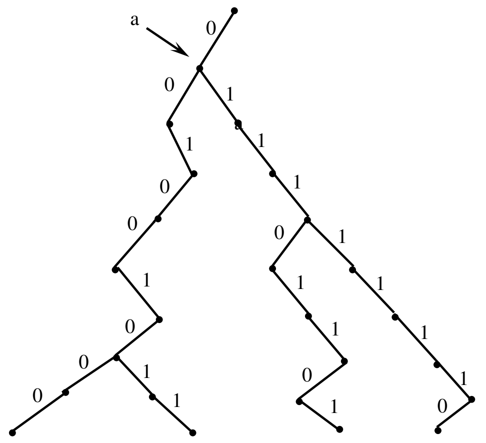

Fig. 1.

In the first pass we do ordered partition of T'_1 by S'_1. Assume that this is done. Let t' be an integer in T'_1 and $s'_i \leq t' \leq s'_{i+1}$, s'_i and s'_{i+1} be the two integers in S'_1 with ranks i and $i+1$. If $t' = s'_i$ then we compare t and $H^{-1}(s'_i)$. Otherwise let the most significant bit that s'_i (s'_{i+1}) and t' differs be the d_i-th (d_{i+1}-th) bit and consider the case that $d_i < d_{i+1}$ (the situation that $d_i > d_{i+1}$ can be treated similarly) in which we will compare t and $H^{-1}(s'_i)$. Note that the situation $d_i = d_{i+1}$ cannot happen. Let d_i-th bit of s'_i be in the g-th segment of s'_i (each segment of s'_i has $\leq 2\log n$ bits and is obtained by hashing the corresponding segment of $2p^{2/3}\log n$ bits in $H^{-1}(s'_i)$. Note here s'_i is not necessarily hashed from s_i because the rank order of s_i's are not kept in s'_i's after hashing, s_i need not be $H^{-1}(s'_i)$, i.e. s'_i is ranked i-th in S'_1 but $H^{-1}(s'_i)$ may not be ranked i-th in S). This says that in the more significant segments than the g-th segment both t' and s'_i are equal. Note that for every s'_j in S'_1 let the most significant bit that t' and s'_j differs be in the g'-th segment then $g' \geq g$ because $s'_i \leq t' \leq s'_{i+1}$.

If we look at the trie (binary tree built following the bits in the integers) for S we see that the most significant segment that t and $H^{-1}(s'_i)$ differs is the u-th segment with $u \geq g$. Note here that u may be not equal to g because H is not perfect for $T \cup S$. For all other s_k's if the most significant segment t and s_k differ is the u_1-th segment then $u_1 \geq u$ (u_1-th segment is at least as significant as u-th segment) and this comes from the property that H is perfect for S. Thus the further ordered partition for t by S can be restricted to the u-th segment of t with respect to the u-th segment of s_k's such that the most significant segment that s_k and t differ is the u-th segment.

Example 1: Fig. 1. shows a trie for U (representing S) of 4 integers: $u_0 = 001001000, u_1 = 001001011, u_2 = 011101101, u_3 = 011111110$. Let set V (representing T) contain integers $v_0 = 000100111, v_1 = 010110011, v_2 = 011011101, v_3 = 010101011, v_4 = 011110101, v_5 = 011111111, v_6 = 001001010, v_7 = 011101011, v_8 = 011010100, v_9 = 011101111$. Let every 3 bits form a segment so that each integer has 3 segments. We will do ordered partition of V by U. Let $w_k = \min_{i=0}^{3}\{w \mid$ the most significant segment that $H(v_k)$ differs with $H(u_i)$ is the w-th segment $\}$. Let the u_i that achieves the w_k value be u_{v_k}.

Say that $w_0 = 0$ and u_{v_0} is u_0. Note that although the most significant segment that u_0 and v_0 differ is the 2nd segment, the most significant segment that $H(u_0)$ and $H(v_0)$ differ can be the 0th segment because H is not perfect on $T \cup S$. Now compare u_0 and v_0 we find that the most significant segment they differ is the 2nd segment. This says that v_0 "branches out" of the trie for U at the 2nd segment. Although the ordered partition for v_0 can be determined from the trie of U, this is not always the case for every integer in V. For example, say $w_3 = 1$ and u_{v_3} is u_0, then comparing u_0 and v_3 determines that v_3 branches out of u_0 at point a in Fig. 1., but we do not know the point where v_3 branches out of the trie for U. Because the 2nd segment of v_3 is different than the 2nd segment of any u_i's and thus the second segment of $H(v_3)$ can be equal to (or not equal to) any of that of $H(u_i)$. The further ordered partition of v_0 and v_3 by U can be restricted to the 2nd segment. $\square$

In the trie for S we will say the g-th segment $s_{i,g}$ of $s_i \in S$ is nonempty if there is an $t \in T$ such that the most significant segment that t and s_i differ is in the g-th segment. If this happens then we will take the prefix $s_{i,prefix} = s_{i,p^{1/3}/2-1}s_{i,p^{1/3}/2-2}...s_{i,g+1}$, i.e. all segments more significant than the g-th segment of s_i. Because there are n integers in T and thus there are at most n nonempty segments. Thus we have at most n such $s_{i,prefix}$ prefix values. We will arbitrarily replace each such prefix value with a distinct number in $\{0, 1, ..., n-1\}$. Say $s_{i,prefix}$ is replaced by $R(s_{i,prefix}) \in \{0, 1, ..., n-1\}$ we then append the g-th segment of t to $R(s_{i,prefix})$ to form $t(1)$ of $\log n + 2p^{2/3} \log n$ bits, where $2p^{2/3} \log n$ bits come from the g-th segment of t and the other $\log n$ bits come from $R(s_{i,prefix})$. s_i is also transformed to $s_i(1)$ by appending the g-th segment of s_i to $R(s_{i,prefix})$. When all the n integers in T and s integers in S are thus transformed to form the sets $T(1)$ and $S(1)$ we left with the problem of ordered partition of $T(1)$ by $S(1)$ and here integers have only $\log n + 2p^{2/3} \log n$ bits.

Example 2: Follows Example 1 and Fig. 1. Because of v_0, v_1, v_3, the 2nd segment of u_0, u_1, u_2, u_3 are not empty. These segments are 001 and 011. The $s_{i,prefix}$ is ϵ. We may take $R(s_{i,prefix}) = 000$ for these two segments (case 0). Because of v_2, v_4, v_8, the 1st segments of u_2, u_3 are not empty. These segments are 101 and 111. The $s_{i,prefix}$ is 011 (the 2nd segment of u_2 and u_3). We may take $R(s_{i,prefix}) = 001$ for them (case 1). Because of v_5, the 0th segment of u_3 is not empty. This segment is 110. The $s_{i,prefix}$ is 011111. We may take $R(s_{i,prefix}) = 010$ for it (case 2). Because of v_6, the 0th segments of u_0 and u_1 are not empty. These segments are 000 and 011. The $s_{i,prefix}$ is 001001. We may take $R(s_{i,prefix}) = 011$ for these segments (case 3). Because of v_7 and v_9, the 0th segment of u_2 is not empty. This segment is 101. The $s_{i,prefix}$ is 0111101. We may take $R(s_{i,prefix}) = 100$ (case 4).

Thus V' contains 000000 (from v_0), 000010 (from v_1), 000010 (from v_3), 001011 (from v_2), 001110 (from v_4), 001010(from v_8), 010111 (from v_5), 011010 (from v_6), 100011 (from v_7), 100111 (from v_9).

U' contains 000001, 000011 (from case 0), 001101, 001111 (from case 1), 010110 (from case 2), 011000, 011011 (from case 3), 100101 (from case 4). $\square$

Thus after one pass of ordered partitioning of a set of n $p^{1/3} \log n$ bit integers (T_1') by a set of $|S|$ $p^{1/3} \log n$ bit integers (S_1'), the original problem of ordered partitioning of T by S is transformed to the problem of ordered partitioning of $T(1)$ by $S(1)$.

If we iterate this process by executing one more pass of ordered partitioning of a set of n $p^{1/3} \log n$ bit integers (call it T_2') by a set of $|S|$ $p^{1/3} \log n$ bit integers (call it S_2') then the ordered partition of $T(1)$ by $S(1)$ will be transformed to the problem of ordered partitioning of $T(2)$ by $S(2)$ where integers in $T(2)$ and $S(2)$ have $2 \log n + 4p^{1/3} \log n$ bits. This basically demonstrates the idea of converting the ordered partition of $p \log n$ bit integers to constant passes of the ordered partition of $p^{1/3} \log n$ bit integers. This idea is not completely new. It can be traced back to [1] where it was named (randomized) signature sorting and used for randomized integer sorting. Here we converted it for deterministic ordered partition.

The principle explained in this subsection give us the follow lemma.

Lemma $2'$**:** Ordered partition of set T by set S with integers of $p \log n$ bits can be converted to ordered partition of set T_i' by S_i' with integers of $p^{1/c_1} \log n$ bits, $i = 1, 2, ..., c_2$, where c_1 and $c_2 = O(c_1)$ are constants. $\square$

3.3 Nonconservative Integer Sorting

An integer sorting algorithm sorting integers in $\{0, 1, ..., m-1\}$ is a nonconservative algorithm [12] if the word size (the number of bits in a word) is $p(\log m + \log n)$, where p is the nonconservative advantage.

Lemma 4 [9, 10]: n integers can be sorted in linear time with a nonconservative sorting algorithm with nonconservative advantage $\log n$. $\square$

4 Ordered Partition of T by S

Now let us consider the problem of the ordered partition of T by $S = \{s_0, s_1, ..., s_{|S|-1}\}$ where $s_i \leq s_{i+1}$ and each integer in T or S has $p \log n$ bits and the word size is $p^3 \log n$ bits. We here consider the case that $|T| = n$ and $|S| = n^{1/8}$. In Sections 2 it is explained why we pick the size of S to be $n^{1/8}$.

First consider the case where $p \geq \log n$. Here we have $p^2 > \log n$ nonconservative advantage and therefore we can simply sort $T \cup S$ using Lemma 4. Thus we assume that $p < \log n$.

Because $|S| = n^{1/8}$ when we finish ordered partition T will be partitioned into $T_0 \leq T_1 \leq \cdots \leq T_{n^{1/8}}$ with $T_0 \leq s_0 \leq T_1 \leq \cdots \leq s_{|S|-1} \leq T_{|S|}$. Let us have a virtual view of the ordered partitioning of T by S. We can view as each integer t in T using $(1/8) \log n$ binary search steps to find the s_i such $s_i \leq t \leq s_{i+1}$ (i.e. in which set T_i it belongs to). The reason it takes $(1/8) \log n$ binary search steps is because we do a binary search on $n^{1/8}$ ordered numbers in S.

As we are partitioning integers in T by integers in S we say that an integer t in T is advanced k steps if t is in the status of having performed $k \log n/p$ binary search steps. Notice the difference when we use word "step" and the phrase "binary search step". Thus each integer needs to advance $p/8$ steps to finish in a set T_i.

Let $v(k) = n^{1/8}/2^{k \log n/p}$ and let $S(k) = \{s_{v(k)}, s_{2v(k)}, s_{3v(k)}, ...\}$, $k = 1, ..., p/8$. Thus $|S(k)| = 2^{k \log n/p}$. Also let $S[k, i] = \{s_{iv(k)}, s_{iv(k)+1}, s_{iv(k)+2}, ..., s_{(i+1)v(k)-1}\}$. If integer t in T is advanced k steps, then we know the i such that $\min S[k, i] \leq t \leq \max S[k, i]$. We will say that t falls into $S[k, i]$. Let $F(S[k, i])$ be the set of integers that fall into $S[k, i]$. When integers in T fall into different $S[k, i]$'s we will continue ordered partition for an $F(S[k, i])$ by $S[k, i]$ at a time. That is, the original ordered partition of T by S becomes multiple ordered partition problems.

Initially we pack p integers into a word. Because each word has size $p^3 \log n$ bits we have p nonconservative advantage after we packed p integers into a word (i.e. we used $p^2 \log n$ bits by packing and now $(\textit{word size})/(p^2 \log n) = p^3 \log n/(p^2 \log n) = p$).

The basic thoughts of our algorithm is as follows: As will be seen that advance n integers by a steps will result in time complexity $O(na/p)$ because when we try to advance a steps we will compare integers in T to integers in $S(a)$ and not to integers in S (the time $O(na/p)$ will be understood when we explain our algorithm). However, it is not that we can directly advance $p/8$ steps and get $O(n)$ time. There is a problem here. If we form one segment for each integer in T (as we did form $p^{1/3}/2$ segments in the previous section) then if the hash value of an integer t in T does not match the hash value of any integer in $S(a)$ then we lose our work because it does not provide any information as to which set T_i t will belong to. If we make multiple segments for each integer t in T then after hashing and comparing t to s_i's we can eliminate all segments except one for t as we did in the previous section. However, it will be understood that using multiple segments will increase the number of bits for the hashed value and this will result in a nonlinear time algorithm for us (the exact details of this will be understood when our algorithm is explained and understood). What we will do is to compare the most significant $\log n$ bits of the integers in T and in S. If the most significant $\log n$ bits of $t \in T$ is not equal to the most significant $\log n$ bits of any integer in S (we call this case as unmatch) then we win because we need to keep only the most significant $\log n$ bits of t and throw away the remaining bits of t as in the trie for S t branches out within the most significant $\log n$ bits. However, if the value of the most significant $\log n$ bits of t is equal to the value of the most significant $\log n$ bits of an integer in S (we call this case as match) then we can only eliminate the most significant $\log n$ bits of t. As t has $p \log n$ bits it will take $O(p)$ steps to eliminate all bits of t and this will result in a nonlinear time algorithm.

What we do is take the most significant $\log n$ bits of every integer and call it the current segment of the integer. The current segment of $a \in T \cup S$ is denoted by $c(a)$. We first advance 1 step for the current segments of T with respect to the current segments of S in time $O(n/p)$. If within this step we find a match for $c(t)$ for $t \in T$ then we eliminate $\log n$ bits of t and make the next $\log n$ bits as the current segment. Note that for $p \log n$ bits this will result in $O(n)$ time as removing $\log n$ bits takes $O(n/p)$ time. If we find unmatch for $c(t)$ then we advance 2 steps with time $O(2n/p)$. If we then find a match for $c(t)$ we then eliminate $\log n$ bits and make the next $\log n$ bits as the current segment. If unmatch we then advance 4 steps in time $O(4n/p)$. We keep doing this then if in the 1st, 2nd, ..., $(a-1)$-th passes we find unmatch for $c(t)$ and in a-th pass we first find a match for $c(t)$ then $c(t)$ matched the $c(s)$ for an $s \in S(2^{a-1})$ and did not match the $c(s)$ for any $s \in S(2^j)$ with $j < a-1$. Examining the structure of $S(2^{a-1})$ and $S(2^{a-2})$ will tell us that $c(t)$ has advanced 2^{a-2} steps (and therefore t advanced 2^{a-2} steps) because we can tell the two integers s_{k_1} and s_{k_2} with ranks k and $k+1$ in $S(2^{a-2})$ such that $s_{k_1} \le t \le s_{k_2}$ because $c(t)$ matches $c(s_i)$ for an s_i in $S(2^{a-1})$ while $S(2^{a-2}) \subset S(2^{a-1})$. The time we spent computing the 1st, 2nd, ..., a-th passes is $O(n2^a/p)$. Thus we advance 2^{a-2} steps by spending time $O(n2^a/p)$. Because we can advance only $p/8$ steps thus the time is linear. If for all passes from pass 1 to pass $\log(p/8)$ we do not find a match for $c(t)$ then we keep the most significant $\log n$ bits of t and remove other bits of t and the time we spent for all these passes is $O(n)$.

The way for us to find whether $c(t)$ matches any $c(s_i)$ is that we first find whether $H(c(t))$ matches any of $H(c(s_i))$ (here we are comparing $H(c(t))$ to all $H(c(s_i))$'s). If not then unmatch, if $H(c(t)) = H(c(s_i))$ then we compare $c(t)$ and $c(s_i)$ (here we are comparing $c(t)$ to a single $c(s_i)$) to find out whether $c(t) = c(s_i)$. Here we need to do this comparison because H is not perfect for the current segment of $T \cup S$. The reason we first compare $H(c(t))$ and all $H(c(s_i))$'s (we actually sort them) is because hashed values contain less number of bits and therefore can be compared more efficiently.

In our algorithm we need to prepend $\log n$ more significant bits (their value is equal to 0 for the first pass) to the $\log n$ bits (the current segment) for every integer we have mentioned above. This is because, say, for $c(t)$ we find unmatch for the first $a-1$ passes and find a match in the a-th pass ($c(t) = c(s_i)$), we then need to remove the most significant $\log n$ bits from t. As we did in the previous section we need to take the $s_{i,prefix}$ and replace it by a distinct integer $R(s_{i,prefix})$ in $\{0, 1, ..., n-1\}$ and then transform t by removing the segments of t that are not less significant than the current segment of t and replace it with $R(s_{i,prefix})$. This is significant as we go down the road as we can repeatedly removing $\log n$ bits (should be $2\log n$ bits after prepending $R(s_{i,prefix})$) from t. The $R(s_{i,prefix})$ value essentially tells on which part of the trie for S we are currently doing ordered partitioning for t. Note that we will do ordered partitioning for all integers fall into an $S[k,i]$ at a time. The prepending of $\log n$ bits is to tell apart of the different $s_{i,prefix}$ values for integers fall into the same $S[k,i]$.

If we have advanced $t \in T$ k steps, then we know the i such that $\min S[k,i] \leq t \leq \max S[k,i]$ and we need to continue to do ordered partition for t by $S[k,i]$. However, different integers may advance at different speed. If integer t is advanced k steps we will use $A(t) = k$ to denote about this. Let $T^{(k)}$ be the set of integers in T that have advanced k steps. $T^{(k)} = \cup_i F(S[k,i])$, where $F(S[k,i])$ are the integers in T that fall into $S[k,i]$.

Consider the case where $F(S[m,m_1])$ is selected for advancement. We will try to advance integers in $F(S[m,m_1])$ for 1 step. Let $T' = F(S[m,m_1])$.

```
Advance for One Step ($T'$)
$U = T'$; $U' = \phi$;
while $|U| \geq n^{1/2}$ do
begin
  1. Try to advance integers in $U$ for 1 step by comparing them with $S[m,m_1] \cap S(m+1)$.
  2. Let $U_{match}$ be the set of matched integers and $U_{unmatch}$ be the set of unmatched integers.
  3. Let $U'_{match}$ be the subset of $U_{match}$ such that the current segment is the least significant segment (i.e. $t \in U'_{match}$ is equal to some $s_i$ in $S$). Integers in $U'_{match}$ are done.
  4. $U = U_{match} - U'_{match}$; $U' = U' \cup U_{unmatch}$;
  5. For $t \in U$ if $c(t)$ matches $c(s_i)$ remove the $c(t)$ from $t$ and make the next $\log n$ bits as the current segment for $t$. Prepend $R(s_{i,prefix})$ to $c(t)$.
end
/* Now $|U| < n^{1/2}$. */
Use brute force to sort integers in $U$;
$T' = U'$;
```

The technique to have **Advance for One Step** done in $O(n/p)$ time is the same as that used in Steps 3.1. and 3.2. in algorithm **Partition** (set $k = 1$ there) shown later and therefore we do not repeat them here.

After we have done algorithm **Advance for One Step** we know that integers in T' will advance at least 1 step later. We then try to advance integers in T' for 2^j steps, $j = 1, 2, 3, ...$ As we do this matched integers will be distributed to $T^{(m+2^{j-1})}$'s. The number of unmatched integers in T' will decrease. When $|T'| < n^{1/2}$ we will use brute force to sort all integers in T'. Thus for integers remain in $F(S[m,m_1])$ they will be sorted by brute force at most once. There are no more than $|S|^2 \leq n^{1/4}$ $S[k,i]$'s. Thus the total number of integers sorted by brute force is no more than $n^{1/2} \cdot n^{1/4} = O(n^{3/4})$. The reason we use brute force to sort them is that the (approximate) ratio of $|T|$ versus $|S|$ as n to $n^{1/8}$ may not hold any longer.

Now our algorithm is shown as follows.

```
Partition($T$, $S$)
/* Partition $T$ by $S$, where integers are in $\{0, 1, ..., 2^{p \log n} - 1\}$ and word size is $p^3 \log n$ bits. */
Initialization: Let current segment of each integer in $T$ or $S$ be the most significant $\log n$ bits of the integer.
For $a \in T \cup S$ let $c(a)$ be its current segment. Label all integers as undone. $T^{(0)} = T$, $T^{(i)} = \phi$ for $i \neq 0$. /* $T^{(i)}$ is the set of integers in $T$ that has advanced $i$ steps. */
1. for($m = 0; m < p/8; m = m + 1$) do Steps 2 to 4.
```

2. For all $S[m,j]$'s such that there are less than $n^{1/2}$ integers in $F(S[m,j])$ sort these integers and mark these integers done (advanced $p/8$ steps). For every m_1 such that there are at least $n^{1/2}$ integers in $F(S[m,m_1])$, let $T^{(m)} = T^{(m)} - F(S[m,m_1])$ and $T' = F(S[m,m_1])$ and let $S' = S[m,m_1]$, do steps 3 through 4. If $T^{(m)}$ is empty do next iteration of Step 1.
3. Call **Advance for One Step** (T'); $k=2$
4. Let $S'(k) = S' \cap S(m+k)$. /* $|S'(k)| = 2^{k \log n/p}$. */

4.1 Use a hash function H to hash the current segment of each integer in $S'(k)$ into $2k \log n/p$ bits. H must be a perfect hash function for the current segments of $S'(k)$. Since $|S'(k)| = 2^{k \log n/p}$ such a hash function can be found [5, 13]. The current segment of each integer in T' is also hashed by this hash function into $2k \log n/p$ bits (here note that there might be collisions as H is not perfect for the current segments of T'). Pack every $p/(8k)$ integers in T' into a word to form a set T'' of words. We also form a set S'' of $n^{1/4}$ words for $S'(k)$. The i-th word W_i in S'' has $p/(8k)$ integers $s_{i_0}, s_{i_1}, ..., s_{i_{p/(8k)-1}}$ in $S'(k)$ packed in it. Here $s_{i_0}, s_{i_1}, ..., s_{i_{p/(8k)-1}}$ must satisfy $H(s_{i_0})H(s_{i_1})...H(s_{i_{p/(8k)-1}}) = i$ (here $H(s_{i_0})H(s_{i_1})...H(s_{i_{p/(8k)-1}})$ is (the value of) the concatenation of the bits in $H(s_{i_j})$'s). Note that because we hash m integers in M to m integers in $M' = \{0, 1, ..., m^2\}$ there are some integers $a' \in M'$ such that no integer a in M is hashed to a'. Thus for some i and j we find that s_{i_j} does not exist in $S'(k)$. If this happens we leave the spot for $s_{i,j}$ in W_i blank, i.e. in this spot no integer in $S'(k)$ is packed into W_i. Note that hashing for integers in T' happens after packing them into words (due to time consideration) and hashing for integers in $S'(k)$ happens before packing as we have to satisfy $H(s_{i_0})H(s_{i_1})...H(s_{i_{p/(8k)-1}}) = i$.

4.2. Since each word of T'' has $p/(8k)$ integers in T' packed in it it has only a total of $\log n/4$ hashed bits because each integer has hashed to $2k \log n/p$ bits. The $n^{1/4}$ words in S'' for packed integers in $S'(k)$ also have $\log n/4$ hashed bits in each of them. We treat these $\log n/4$ bits as one INTEGER and sort T'' and S'' together using the INTEGER of them as the sorting key. This sorting can be done by bucket sort (because INTEGER is the sorting key) in linear time in terms of the number of words or in $O(|T'|k/p)$ time (because we packed $p/(8k)$ integers in one word). This sorting will bring all words with same INTEGER key together. Because S'' has word with INTEGER keys with any value in $\{0, 1, ..., n^{1/4}\}$ any word w in T'' can find w_1 in S'' with the same INTEGER key value and thus w will be sorted together with w_1. Thus now each $c(t_i)$ with $t_i \in T'$ can find that either there is no $c(s_j)$ with $s_j \in S'(k)$ such that $H(c(t_i)) = H(c(s_j))$ (in this case $c(t_i) \neq c(s_j)$ for any $s_j \in S'(k)$) or there is an s_j such that $H(c(t_i)) = H(c(s_j))$. If $H(c(t_i)) = H(c(s_j))$ then we compare $c(t_i)$ with $c(s_j)$ (we need to do this comparison because H is not perfect for $T' \cup S'(k)$). If they are equal (match) then if the current segment is not the least significant segment then we eliminate the current segment of t_i (in this case we find all segments that are not less significant than the current segment of t_i are equal to the corresponding segments of s_j) and mark the next $\log n$ bits as its current segment. Also eliminate the current segment of s_j and use the next $\log n$ bits of s_j as its current segment (the idea here is similar to the mechanism in the previous section where we eliminates the preceding segments of t_i and s_j because they are equal). If the current segment before removing is the least significant segment then $t_i = s_j$ and we are done with t_i. Let D be the set of integers t_i in T' such that $c(t_i) = c(s_j)$ for some $s_j \in S'(k)$. Advance all integers in D $k/2$ steps (because they matched in $S'(k)$ and did not match in $S'(k/2)$) and let $T' = T' - D$, $T^{(m+k/2)} = T^{(m+k/2)} \cup D$. For integers in D we can also determine the $S[m+k/2, l]$ on which t_i falls into. For integers in D remove the current segment of these integers. For integers in D that has been distributed into $F(S[m+k/2,l])$'s we have to prepend $R(s_{j,prefix})$ to $c(t_i)$. T' is now the set of unmatched integers.

Note here we have to separate integers in D from integers not in D. For integers in D we have to further advance them later according to which $S[m+k/2,l]$ they fall into. This is done by labeling integers not in D with 0 and integers fall into $S[m+k/2,l]$ with $l+1$. The labels form an integer LABEL (just as we formed INTEGER before) for integers packed in a word. We then use bucket sort to sort all words using LABEL as the key. This will bring all words with the same LABEL value together. For every p words with the same LABEL value we do a transpose (i.e. move the i-th integer (LABEL) in these words into a new word). This transposition would take $O(p\log p)$ time for p words if we do it directly. However we can first pack $\log p$ words into one word (because word size is $p^3\log n$) and then do the transposition. This will bring the time for transposition to $O(p)$ for p words. This will have integers separated.

4.3. If $|T'| < n^{1/2}$ then use brute force to sort all integers in T' and mark them as done (advanced to step $p/8$). Goto Step 2.

4.4. If $m+k = p/8$ then mark all integers in T' as done (advanced to step $p/8$). keep the current segment of them and remove all other segments of them. Let $E(m, m_1) = T'$. Goto Step 2.

4.5. If $p/8 < m+2k$ let $k = p/8 - m$ else let $k = 2k$ and goto Step 4.

5. We have to sort integers (of $2\log n$ bits) in every nonempty set $E(m, m_1)$. Here each integer in $E(m, m_1)$ has its current segment differ from the current segment of any $s \in S[m, m_1]$. We mix them together and sort them by radix sort. Then we go from the smallest integer to the largest integer one by one and separate them by their index (m, m_1). This will have every set $E(m, m_1)$ sorted. This case corresponds to unmatch for all passes and we end up with $2\log n$ bits for each of these integers.

5 Ordered Partition

We have explained in Section 2 how the ordered partition is done using the results in Sections 3 and 4 (i.e. how ordered partition is done by ordered partition of T by S). Therefore we have our main theorem:

Main Theorem: Ordered partition of a set T of n integers into $n^{1/2}$ sets $T_0 \le T_1 \le \cdots \le T_{n^{1/2}-1}$, where $|T_i| = \theta(n^{1/2})$ and $T_i \le T_{i+1}$ means that $\max T_i \le \min T_{i+1}$, can be computed in linear time and space. $\square$

6 Randomization and Nonconservativeness

Although our ordered partition algorithm is deterministic and conservative (i.e. use word of size $\log m + \log n$ for integers in $\{0, 1, ..., m-1\}$), it may be used in the randomized setting or the nonconservative setting. Here we consider the situation under these settings.

In randomized setting n integers with word size $\Omega(\log^2 n \log\log n)$ can be sorted in linear time [3] and thus ordered partition is not needed. Han and Thorup have presented a randomized integer sorting algorithm with time $O(n(\log\log n)^{1/2})$ [11] and it represents a tradeoff between ordered partition and sorting.

In [7] the schemes in this paper is extended and combined with other ideas to obtain a randomized linear time algorithm for sorting integers with word size $\Omega(\log^2 n)$, i.e. integers larger than $2^{\log^2 n}$, thus improve the results in [3].

7 Conclusion

We hope that our ordered partition algorithm will help in the search for a linear time algorithm for integer sorting. We tend to believe that integers can be sorted in linear time, as least in the randomized setting. There are still obstacles to overcome before we can achieve this goal.

Acknowledgement

Reviewers have given us very helpful comments and suggestions which helped us improve the presentation of this paper significantly. We very much appreciate their careful reviewing work.

References

1. A. Andersson, T. Hagerup, S. Nilsson, R. Raman. Sorting in linear time? *Journal of Computer and System Science* **57**, 74-93(1998).
2. A, Andersson. Faster deterministic sorting and searching in linear space. *Proc. 1996 IEEE Symposium of Foundations of Computer Science FOCS'1996*, 135-141(1996).
3. D. Belazzougui, G. S. Brodal, J. S. Nielsen. Expected linear time sorting for word size $\Omega(\log^2 n \log\log n)$. In SWAT'2014. LNCS 8503, 26-37(2014).
4. M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, R. E. Tarjan. Time bounds for selection. *J. Computer and System Sciences*, **7:**4, 448-461(1972).
5. M. Dietzfelbinger, T. Hagerup, J. Katajainen, M. Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms* **25**, 19-51(1997).
6. Y. Han. Deterministic sorting in $O(n \log\log n)$ time and linear space. *Journal of Algorithms*, 50, 96-105(2004).
7. Y. Han. Optimal randomized integer sorting for integers of word size $\Omega(\log^2 n)$. Manuscript.
8. Y. Han. Construct a perfect hash function in time independent of the size of integers. Manuscript.
9. Y. Han, X. Shen. Conservative algorithms for parallel and sequential integer sorting. *Proc. 1995 International Computing and Combinatorics Conference, Lecture Notes in Computer Science* **959**, 324-333(1995).
10. Y. Han, X. Shen. Parallel integer sorting is more efficient than parallel comparison sorting on exclusive write PRAMs. *SIAM J. Comput.* 31, 6, 1852-1878(2002).
11. Y. Han, M. Thorup. Integer Sorting in $O(n\sqrt{\log\log n})$ Expected Time and Linear Space, *Proceedings of the 43rd IEEE Symposium on Foundations of Computer Science(FOCS'02)*, 135–144(2002).
12. D. Kirkpatrick, S. Reisch. Upper bounds for sorting integers on random access machines. *Theoretical Computer Science* 28, 263-276(1984).
13. R. Raman. Priority queues: small, monotone and trans-dichotomous. *Proc. 1996 European Symp. on Algorithms, Lecture Notes in Computer Science 1136*, 121-137(1996).