Deterministic Sorting in $O(n \log\log n)$ Time and Linear Space[1]

Yijie Han

School of Interdisciplinary Computing and Engineering
University of Missouri at Kansas City
5100 Rockhill Road
Kansas City, MO 64110
hanyij@umkc.edu
http://welcome.to/yijiehan

Abstract

We present a fast deterministic algorithm for integer sorting in linear space. Our algorithm sorts n integers in the range $\{0, 1, 2, ..., m-1\}$ in linear space in $O(n \log\log n)$ time. This improves our previous result[8] which sorts in $O(n \log\log n \log\log\log n)$ time and linear space. This also improves previous best deterministic sorting algorithm[3, 11] which sorts in $O(n \log\log n)$ time but uses $O(m^\epsilon)$ space. Our results can also be compared with Thorup's previous result[16] which sorts in $O(n \log\log n)$ time and linear space but uses randomization.
Keywords: Algorithms, sorting, integer sorting, time complexity, linear space.

1 Introduction

Sorting is a classical problem which has been studied by many researchers. Although the complexity for comparison sorting is now well understood, the picture for integer sorting is still not clear. The only known lower bound for integer sorting is the trivial $\Omega(n)$ bound. Continuous research efforts have been made by many researchers on integer sorting[2, 3, 6, 7, 8, 9, 11, 12, 14, 15, 16]. Recent advances in the design of algorithms for integers sorting have resulted in fast algorithms[3, 11, 16]. However, these algorithms use randomization or superlinear space. For sorting integers in $\{0, 1, ..., m-1\}$ $O(m^\epsilon)$ space is used in the algorithms reported in [3, 11]. When m is large (say $m = \Omega(2^n)$) the space used is excessive. Integer sorting using linear space is therefore extensively studied by researchers. An earlier work by Fredman and Willard[6] shows that n integers can be sorted in $O(n \log n / \log\log n)$ time in linear space. Raman[14] showed that sorting can be done in $O(n\sqrt{\log n \log\log n})$ time in linear space. Later Andersson[2] improved the time bound to $O(n\sqrt{\log n})$. Then Thorup[15] improved the time bound to $O(n(\log\log n)^2)$. Our previous results showed time $O(n(\log\log n)^{3/2})$ [9] and the previous best result of $O(n \log\log n \log\log\log n)$ [8]. In this

[1]Preliminary version of this paper has been presented at 2002 ACM Symposium on Theory of Computing (STOC'02).

paper we further improve upon previous results. We show that n integers in $\{0, 1, 2, ..., m-1\}$ can be sorted in $O(n \log\log n)$ time in linear space.

Our result improves on time on the previous best linear space sorting algorithm[8] which uses $O(n \log\log n \log\log\log n)$ time. Our result also improves on space on the previous fastest deterministic sorting algorithm[3, 11] which sorts in $O(n \log\log n)$ time and $O(m^\epsilon)$ space, where $\{0, 1, ..., m-1\}$ is the range of the integers. This previous result was obtained independently by Andersson *et al.* [3] and by Han and Shen[11]. The space used in these previous algorithms is actually $O(m)$. But we may assume that space is reduced to $O(m^\epsilon)$ by using radix sorting. Our result can also be compared with Thorup's result[16] which sorts in $O(n \log\log n)$ time and linear space using randomization. However, although our algorithm do not use randomization we use multiplication instruction in our algorithm while Thorup's algorithm uses randomization but without using multiplication instruction.

The techniques used in our algorithm include coordinated pass down of integers on the Andersson's exponential search tree[2] and the linear time multi-dividing of the bits of integers. Although we used multi-dividing technique in our previous design[8], there multi-dividing takes nonlinear time and therefore is too slow. Our new multi-dividing can only be accomplished with coordinated pass down of integers. Instead of inserting integers one at a time into the exponential search tree we pass down all integers one level of the exponential search tree at a time. Such coordinated passing down provides us the chance of performing multi-dividing in linear time and therefore speeding up our algorithm.

We would like to comment on the complexity of $O(n \log\log n)$. This bound was manifested as the best bound even for non-linear space deterministic sorting. Andersson [2] showed several algorithms for sorting, none of them could break the $O(n \log\log n)$ bound. Even for very large integers Andersson showed time $O(n(\log n/\log b + \log\log n))$ where b is the word length(the number of bits in a word). Thus no matter how large the integer is $O(n \log\log n)$ time is needed in Andersson's algorithm. In contrast for very large integers its large word length can be exploited in a randomized algorithm[3]. Since Andersson's exponential search tree requires $O(n \log\log n)$ time to balance, it would be unlikely that any deterministic algorithm uses exponential search tree approach could undercut the $O(n \log\log n)$ time complexity. As the time of $O(n \log\log n)$ is the converge point for currently the best bound for linear space sorting as demonstrated in this paper, for non-linear space sorting as shown in [3, 11], and for a randomized linear space sorting [16], it can be viewed as we have reached a milestone.

Although $O(n \log\log n)$ is a natural deterministic bound, recently Han and Thorup find that this complexity can be improved in a randomized setting. In [10] Han and Thorup obtained a randomized integer sorting algorithm which sorts in $O(n\sqrt{\log\log n})$ time and

linear space.

2 Preliminary

Our algorithm is built upon the concept of Andersson's exponential search tree[2]. An exponential search tree of n leaves consists of a root r and n^ϵ exponential search subtrees, $0 < \epsilon < 1$, each having $n^{1-\epsilon}$ leaves and rooted at a child of r. Thus an exponential search tree has $O(\log\log n)$ levels. Sorting is done by inserting integers into the exponential search tree. When imbalance happens in the tree rebalance needs to be done. In [2] Andersson has shown that rebalance takes $O(n\log\log n)$ time when n integers are inserted into the tree. The dominating time is taken by the insertion. Andersson has shown that insertion can be done in $O(\sqrt{\log n})$ time. He inserts one integer into the exponential tree at a time. Thorup[15] finds that by inserting integers in batches the amortized time for insertion can be reduced to $O(\log\log n)$ for each level of the tree. The size of one batch b at a node is defined by Thorup to be equal to the number of children d of the node. In our previous design[8, 9] we pass down d^2 integers in a batch. We showed[8, 9] that we can speed up computation by such a scheme.

An integer sorting algorithm sorts n integers in $\{0, 1, ..., m-1\}$ is called a conservative algorithm[12] if the word length (the number of bits in a word) used in the algorithm is $O(\log(m+n))$. It is called a nonconservative algorithm if the word length used is larger than $O(\log(m+n))$.

One way to speed up sorting is to reduce the number of bits in integers. After the number of bits is reduced we can apply nonconservative sorting. If we are sorting integers in $\{0, 1, ..., m-1\}$ with word length $k\log(m+n)$ with $k \geq 1$ then we say that we are sorting with nonconservative advantage k.

We use the following notation. For a set S we let $\min(S) = \min\{a|a \in S\}$ and $\max(S) = \max\{a|a \in S\}$. For two sets S_1, S_2 we denote $S_1 < S_2$ if $\max(S_1) \leq \min(S_2)$.

One way to reduce the number of bits in an integer is to use bisection (binary dividing) on the bits of the integer (it is sometimes called exponential range reduction). This idea was first invented by van Emde Boas *et al.* [4]. In each step, the number of remaining bits is reduced to half. Thus in $\log\log m$ steps $\log m$ bits of the integers are reduced to constant number of bits. This scheme, although very fast, requires a very large amount of memory. It requires $O(m)$ memory cells and therefore cannot be directly executed in linear space ($O(n)$ space). Andersson[2] invented the exponential search tree and he used perfect hashing to reduce the space to linear. He can store only one integer into a word and then applies the hash function. To speed up the algorithm for sorting, we need to pack several integers into

one word and then to use constant number of steps to accomplish the hashing for all integers stored in the word. In order to achieve this we relax the demand of perfect hashing. We do not demand n integers to be hashed into a table of size $O(n)$ without any collision. A hash function hashes n integers into a table of size $O(n^2)$ in constant time and without collision suffice for us. Therefore we use the improved version of the hashing function given by Dietzfelbinger *et al.* [5] and Raman[14] as shown in the following Lemma.

Let $b \geq 0$ be an integer and let $U = \{0, ..., 2^b - 1\}$. The class $\mathcal{H}_{b,s}$ of hash functions from U to $\{0, ..., 2^s - 1\}$ is defined as $\mathcal{H}_{b,s} = \{h_a | 0 < a < 2^b$, and a is odd $\}$ and for all $x \in U$:

$$h_a(x) = (ax \bmod 2^b) \operatorname{div} 2^{b-s}$$

Lemma 1(Lemma 9 in [14]): Given integer $b \geq s \geq 0$ and $T \subseteq \{0, ..., 2^b - 1\}$ with $|T| = n$, and $t \geq 2^{-s+1} \binom{n}{2}$, a function $h_a \in \mathcal{H}_{b,s}$ can be chosen in $O(n^2 b)$ time such that the number of collisions $\text{coll}(h_a, T) \leq t$.

Take $s = 2 \log n$ we obtain a hash function h_a which hashes n integers in U into a table of size $O(n^2)$ without any collision. Obviously $h_a(x)$ can be computed for any given x in constant time. If we pack several integers into one word and have these integers properly separated with several bits of 0's we can safely apply h_a to the whole word and the result is that hashing values for all integers in the word have been computed. Note that this is possible because only the computation of a multiplication, mod 2^b and div 2^{b-s} is involved in computing a hash value.

Andersson *et al.* [3] used a randomized version of a hash function in $\mathcal{H}$ because they could not afford to construct the function deterministically.

A problem with Raman's hash function is that it takes $O(n^2 b)$ time to find the right hash function. Here b is the number of the bits in an integer. What we needed is a hash function which can be found in $O(n^c)$ time for a constant c because this is needed in the exponential search tree [2, 14]. Obviously Raman's hash function does not satisfy this criterion when b is large. However, Andersson's result[2] says that n integers can be sorted in linear space in $O(n(\log n / \log b + \log \log n))$ time. Thus if $b > n$ we simply use Andersson's sorting algorithm to sort in $O(n \log \log n)$ time. Thus the only situation we have to consider is $b \leq n$. Fortunately for this range of b $O(n^2 b) = O(n^3)$. Therefore we can assume the right hash function can be found in $O(n^3)$ time.

Note that although the hash table has size $O(n^2)$ it does not affect our linear space claim because we do not use hash value to index into a table. Hashing is only used to serve the purpose of reducing the number of bits in an integer.

We will use signature sorting[3] in our algorithm. Signature sorting works as follows.

Suppose that n integers have to be sorted and each integer has $\log m$ bits. We view that each integer has h segments with each segment containing $\log m/h$ bits. Now we apply hashing to each and every segment in each integer and we get $2h \log n$ bits of hashed values for each integer. After sorting on hashed values for all integers the original sorting problem (of sorting n integers of $\log m$ bits each) can be transformed to the sorting problem of sorting n integers of $\log m/h$ bits each.

We will also study the following partitioning problem. Let $a_1, a_2, ..., a_p$ be p integers and S is a set of integers. We intend to partitioning S into $p+1$ sets as $S_0 < \{a_1\} < S_1 < \{a_2\} < ... < \{a_p\} < S_p$. Because we use signature sorting, before we do the above partitioning we will partition the bits in a_i into h segments and take some of these h segments. We will also partition bits for each integer in S into h segment and take one segment and discard other segments. For each a_i we essentially take all the h segments. However, the corresponding segments of a_i and a_j may be identical. In this case we just need one of them. The segment we take for an integer in S is the segment which "branches out" of a_i's. We therefore transform the original partitioning problem into several partitioning problems with integers of $\log m/h$ bits. In Fig. 1 we show that $a_1 = 3$, $a_2 = 5$, $a_3 = 7$, $a_4 = 10$, $S = \{1, 4, 6, 8, 9, 13, 14\}$. We partition each integer into 2 segments. From $a_1 = 3$ we obtain upper segment 0, lower segment 3. From $a_2 = 5$ we obtain upper segment 1 and lower segment 1. From $a_3 = 7$ we obtain upper segment 1 and lower segment 3. From $a_4 = 10$ we obtain upper segment 2 and lower segment 2. For $1 \in S$ we obtain lower segment 1 because it branches out from $a_1 = 3$ in the lower segment. For $4 \in S$ we obtain lower segment 0. For $8 \in S$ we obtain lower segment of 0. For $9 \in S$ we obtain lower segment of 1. For $13 \in S$ we obtain upper segment of 3. For $14 \in S$ we obtain upper segment of 3. Now all upper segments form one new partitioning problem. The lower segments of 1 and 3 Form a new partitioning problem. The lower segments of 4,5,6,7 form a new partitioning problem. The lower segments of 8, 9, 10 form a new partitioning problem. Therefore we now have 4 partitioning problems.

3 Sorting on Small Integers

In this and the next section we will show how the following Lemma 2 is proved. The contents of this and next section have appeared in [8]. We include a modified version of them here for the completeness of this paper.

Lemma 2: n integers can be sorted into $\sqrt{n}$ sets $S_1, S_2, ..., S_{\sqrt{n}}$ such that each set has $\sqrt{n}$ integers and $S_i < S_j$ if $i < j$, in time $O(n \log\log n/\log k)$ and linear space with nonconservative advantage $k \log\log n$.

In integer sorting we often pack several small integers into one word. We always assume

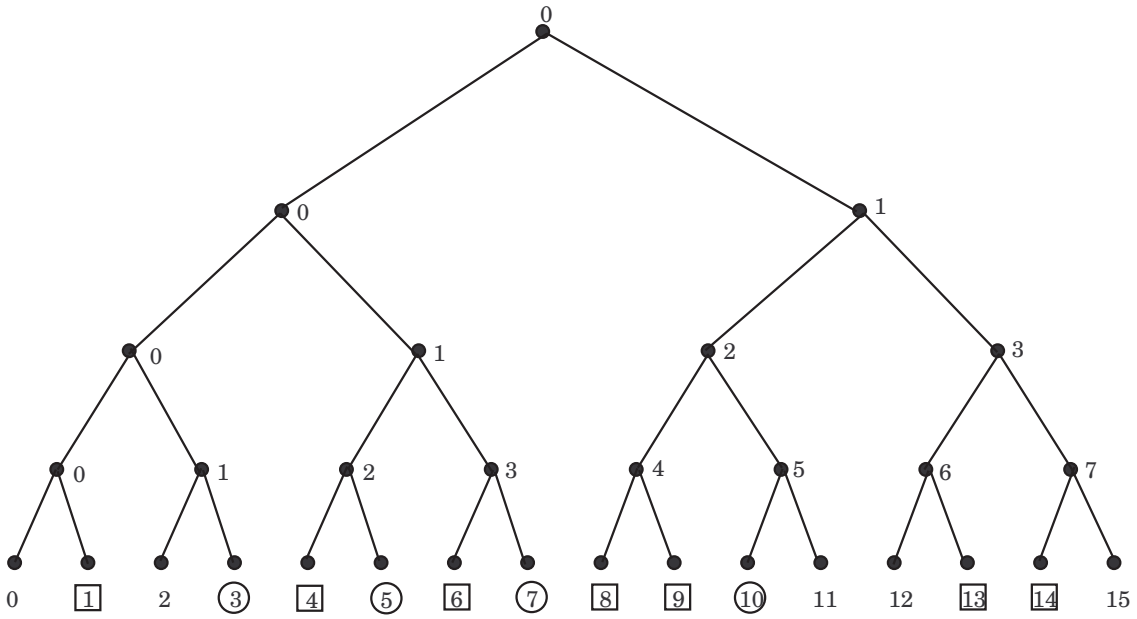

Fig. 1. Set partitioning. The numbers in circles are partitioning integers. The numbers in squares are integers in set S.

that all the integers packed in a word use the same number of bits. Suppose g integers each having l bits are packed into one word. By using the test bit technique [1, 3] we can do a pairwise comparison of the corresponding integers in two words and extract the larger integers into one word and smaller integers into another word in constant time. Therefore by adapting well-known selection algorithms we obtain the following lemma:

Lemma 3: Selecting the s-th largest integer among the n integers packed into n/g words can be done in $O(n \log g/g)$ time and $O(n/g)$ space. In particular the median can be found in $O(n \log g/g)$ time and $O(n/g)$ space.

Proof: Since we can do pairwise comparison of g integers in one word with g integers in another word and extract the larger integers in one word and smaller integers in another word in constant time, we can extract the medians of the 1st, 2nd, ... g-th integer of 5 words into one word in constant time. Thus the set S of medians are now contained in $n/(5g)$ words. Recursively find the median m in S. Use m to eliminate at lease $n/4$ integers among the n integers. Then pack the remaining integers in n/g words into $3n/(4g)$ words (the packing incurs the factor $\log g$ in the time complexity) and then recurse. Packing can be done by the packing algorithm in Leighton[13] (Section 3.4.3). □

Now consider sorting small integers. Let g integers be packed in one word. We say that the ng integers in n words are sorted if gi-th to $(g(i+1)-1)$-th smallest integers are sorted

and packed in the i-th word, $0 \leq i < n$. We have the following lemma:

Lemma 4: If g integers using a total of $(\log n)/2$ bits are packed into one word, then the n integers in n/g words can be sorted in $O((n/g)\log g)$ time and $O(n/g)$ space.

Proof: Because only $(\log n)/2$ bits are used in each word to store g integers we can use bucket sorting to sort all words by treating each word as one integer and this takes $O(n/g)$ time and space. Because only $(\log n)/2$ bits are used in each word there are only $\sqrt{n}$ patterns for all the words. We then put $g < (\log n)/2$ words with the same pattern into one group. For each pattern there are at most $g-1$ words left which cannot form a group. Therefore at most $\sqrt{n} \cdot (g-1)$ words cannot form groups. For each group we move the i-th integer in all g words into one word. That is, we take g g-integer vectors and produce g g-integer vectors where the i's vector contains i-th integer from each input vector. This transpose operation can be done with Lemma 5.4 in Thorup[16] in time $O(g \log g)$ and space $O(g)$. Therefore for all groups it takes $O((n/g)\log g)$ time and $O(n/g)$ space.

For the words not in a group (there are at most $\sqrt{n}\cdot(g-1)$ of them) we simply disassemble the words and then reassemble the words. This will take no more than $O(n/g)$ time and space. After all these are done we then use bucket sorting again to sort the n words. This will have all the integers sorted. □

Note that when $g = O(\log n)$ we are sorting $O(n)$ integers packed in n/g words in $O((n/g)\log\log n)$ time and $O(n/g)$ space. Therefore the saving is considerable.

Lemma 5: Assume that each word has $\log m > \log n$ bits, that g integers each having $(\log m)/g$ bits are packed into one word, that each integer has a label containing $(\log n)/(2g)$ bits, and that the g labels are packed into one word the same way as integers are packed into words (that is, if integer a is packed as the s-th integer in the t-th word then the label for a is packed as the s-th label in the t-th word for labels), then n integers in n/g words can be sorted *by their labels* in $O((n\log\log n)/g)$ time and $O(n/g)$ space.

Proof: The words for labels can be sorted by bucket sorting because each word uses $(\log n)/2$ bits. The sorting will group words for integers into groups as in Lemma 4. We can transpose each group of words for integers. □

Note also that the sorting algorithm given in Lemma 4 and Lemma 5 are not stable. As will be seen that sorting algorithms built on them can be made stable by using the well known method of appending the address bits to each input integer.

If we have larger word length the sorting can be done faster as shown in the following lemma.

Lemma 6: Assume that each word has $\log m \log\log n > \log n$ bits, that g integers each having $(\log m)/g$ bits are packed into one word, that each integer has a label containing $(\log n)/(2g)$ bits, and that the g labels are packed into one word the same way as integers

are packed into words, then n integers in n/g words can be sorted by their labels in $O(n/g)$ time and $O(n/g)$ space.

Proof: Note that although word length is $\log m \log\log n$ only $\log m$ bits are used for storing packed integers. As in Lemmas 4 and 5 we sort the words containing packed labels by bucket sorting. In order to transpose words of integers we put $g\log\log n$ words of integers into one group instead of putting g words of integers into one group. To transpose the integers in a group containing $g\log\log n$ words we first further pack $g\log\log n$ words into g words by packing $\log\log n$ words of integers into one word. We then do transpose on the g words. Thus transpose takes only $O(g\log\log n)$ time for each group and $O(n/g)$ time for all integers. After finishing transpose we then unpack the integers in the g words into $g\log\log n$ words. □

Note also if the word length is $\log m \log\log n$ and only $\log m$ bits are used to pack $g \leq \log n$ integers into one word. Then the selection in Lemma 3 can be done in $O(n/g)$ time and space because the packing in the proof of Lemma 3 can now be done in $O(n/g)$ time.

4 Sort n integers into $\sqrt{n}$ sets

Consider the problem of sorting n integers in $\{0, 1, ..., m-1\}$ into $\sqrt{n}$ sets as given in Lemma 2. We assume that each word has $k\log\log n\log m$ bits and stores an integer of $\log m$ bits. Therefore the nonconservative advantage is $k\log\log n$. We also assume that $\log m \geq \log n \log\log n$. Otherwise we can use radix sorting to sort in $O(n\log\log n)$ time and linear space. We divide the $\log m$ bits used for representing each integer into $\log n$ blocks. Each block thus contains at least $\log\log n$ bits. The i-th block containing $(i\log m/\log n)$-th to $((i+1)\log m/\log n - 1)$-th bits. Bits are counted from the least significant bit starting at 0. We now give a $2\log n$ stage algorithm which works as follows.

In each stage we work on one block of bits. We call these blocks small integers because each small integer now contains only $\log m/\log n$ bits. Each integer is represented by and corresponds to a small integer which we are working on. Consider the 0-th stage which works on the most significant block (the $(\log n - 1)$-th block). Assume that the bits in these small integers are packed into $n/\log n$ words with $\log n$ small integers packed into one word. For the moment we ignore the time needed for packing these small integers into $n/\log n$ words and assume that this is done for free. By Lemma 3 we can find the median of these n small integers in $O(n/\log n)$ time(note that we have at least $\log\log n$ nonconservative advantage) and $O(n/\log n)$ space. Let a be the median found. Then n small integers can be divided into at most three sets S_1, S_2, and S_3. S_1 contains small integers which are less than a. S_2 contains small integers which are equal to a. S_3 contains small integers which are greater

than a. We also have $|S_1| \leq n/2$ and $|S_3| \leq n/2$. Although $|S_2|$ could be larger than $n/2$ all small integers in S_2 are equal. Let S_2' be the set of integers whose most significant block is in S_2. Then we can eliminate $\log m/\log n$ bits (the most significant block) from each integer in S_2' from further consideration. Thus after one stage each integer is either in a set whose size is at most half of the size of the set at the beginning of the stage, or one block of bits ($\log m/\log n$ bits) of the integer can be eliminated from further computation. Because there are only $\log n$ blocks in each integer, each integer takes at most $\log n$ stages to eliminate blocks of bits. An integer can be put in a half sized set for at most $\log n$ times. Therefore after $2\log n$ stages all integers are sorted. Because in each stage we are dealing with only $n/\log n$ words, if we ignore the time needed for packing small integers into words and for moving small integers to the right set then the remaining time complexity will be $O(n)$ because there are only $2\log n$ stages.

The subtle part of the algorithm is how to move small integers into the set where the corresponding integer belongs after previous set dividing operations of our algorithm. Suppose that n integers have already been divided into e sets. We can use $\log e$ bits to label each set. We wish to apply Lemma 6. Since the total label size in each word has to be $\log n/2$, and each label uses $\log e$ bits, the number g of labels in each word has to be at most $\log n/(2\log e)$. Further, since $g = \log n/(2\log e)$ small integers should fit in a word, and each word contains $k\log\log n\log n$ blocks, each small integer can contain $O(k\log n/g) = O(k\log e)$ blocks. Note that we reserve $\log\log n$ nonconservative advantage for the purpose of being used in Lemma 6. Thus we assume that $(\log n)/(2\log e)$ small integers each containing $k\log e$ continuous blocks of an integer are packed into one word. For each small integer we use a label of $\log e$ bits indicating which set it belongs. Assume that the labels are also packed into words the same way as the small integers are packed into words with $(\log n)/(2\log e)$ labels packed into one word. Thus if small integer a is packed as the s-th small integer in the t-th word then the label for a is packed as the s-th label in the t-th word for labels. Note that we cannot disassemble the small integers from the words and then move them because this will incur $O(n)$ time. Because each word for labels contains $(\log n)/(2\log e)$ labels therefore only $(\log n)/2$ bits are used for each such word. Thus Lemma 6 can be applied here to move the small integers into the sets they belong to. Because only $O((n\log e)/\log n)$ words are used the time complexity for moving small integers to their sets is $O((n\log e)/\log n)$.

Note that $O(k\log e)$ blocks for each small integer is the most number of bits we can move in applying Lemma 6 because each word has $k\log\log n\log m$ bits and we want to reserve $\log\log n$ nonconservative advantage. Note also that the moving process is not stable as the sorting algorithm in Lemma 6 is not stable.

With such a moving scheme we immediately face the following problem. If integer a is

the i-th member of a set S. That is, a block of a (call it a') is listed as the i-th (small) integer in S. When we use the above scheme to move the next several blocks of a (call it a'') into S, a'' is merely moved into a position in set S, but not necessarily to the i-th position (the position where a' locates). If the value of the block for a' is identical for all integers in S that does not create problem because that block is identical no matter which position in S a'' is moved to. If the value of the block for a' is not identical for all integers in S then we have problem continuing the sorting process. What we do is the following. At each stage the integers in one set works on a common block which is called the current block of the set. The blocks which precede the current block contain more significant bits of the integer and are identical for all integers in the set. When we are moving more bits into the set we move the following blocks together with the current block into the set. That is, in the above moving process we assume the most significant block among the $k \log e$ continuous blocks is the current block. Thus after we move these $k \log e$ blocks into the set we delete the original current block because we know that the $k \log e$ blocks are moved into the correct set and that where the original current block locates is not important because that current block is contained in the $k \log e$ blocks.

Another problem we would like to mention is that the size of the sets after several stages of dividing will become small. The scheme of Lemmas 4, 5 and 6 relies on the fact that the size of the set is not very small. Since we are sorting a set of size n to sets of size $\sqrt{n}$ we should have no problem. If we want to use our scheme to sort the whole input set we can use a recursion to keep sorting input set into smaller sets. The details of this can be found in [8].

Below is our sorting algorithm which is used to sort integers into sets of size $\sqrt{n}$. This algorithm uses yet another recursion (do not confuse this recursion with the recursion mentioned in the above paragraph).

Algorithm Sort($k \log \log n, level, a_0, a_1, ..., a_t$)
/* $k \log \log n$ is the nonconservative advantage. a_i's are the input integers in a set to be sorted. $level$ is the recursion level. */
1. **if** $level = 1$ **then** examine the size of the set (i.e. t). If the size of the set is less than or equal to $\sqrt{n}$ then return. Otherwise use the current block to divide the set into at most three sets by using Lemma 3 to find the median and then using Lemma 6 to sort. For the set all of its elements are equal to the median eliminate the current block and note the next block to become the current block. Create a label which is the set number (0, 1 or 2 because the set is divided into at most three sets) for each integers. Then reverse the computation to route the label for each integer back to the position where the integer located in the input

to the procedure call. Also route a number (a 2 bit number) for each integer indicating the current block back to the location of the integer. Return.
2.
for $u = 1$ to k do:
begin

2.1. Pack $a_i^{(v)}$'s into a fraction of $1/k$-th of the number of words, where $a_i^{(v)}$ contains several contiguous blocks which consist of $1/k$-th of the bits in a_i and has the current block as its most significant block.

2.2. Call Sort$(k \log\log n, level - 1, a_0^{(v)}, a_1^{(v)}, ..., a_t^{(v)})$. /*When the algorithm returns from this recursive call the label for each integer indicating the set the integer belongs is already routed back to the position where the integer locates in the input of the procedure call. A number having at most the number of bits in a_i indicating the current block in a_i is also routed back to a_i. */

2.3. Route a_i's to their sets by using Lemma 6.

end

Note that when the recursive call at step 2.2. returns the number of eliminated bits in different sets could be different. For the subsequent recursive calls to continue we have to pack $a_i^{(v)}$'s, namely we have to extract a segment which has the current block as its most significant block. Also note that since we have nonconservative advantage k we can move the whole a_i in step 2.3.

We let a block contain $(4 \log m)/\log n$ bits. Then if we call Sort$(k \log\log n, \log_k((\log n)/4), a_0, a_1, ..., a_{n-1})$ where a_i's are the input integers, $(\log n)/4$ calls to the level 1 procedure will be executed. This could split the input set into $3^{(\log n)/4}$ sets. And therefore we need $\log 3^{(\log n)/4}$ bits to represent/index each set. We call Sort several times as below:

Algorithm IterateSort
Call Sort$(k \log\log n, \log_k((\log n)/4), a_0, a_1, ..., a_{n-1})$;
for $j = 1$ **to** 5 **do**
begin
Move a_i to its set by bucket sorting because there are only about $\sqrt{n}$ sets;
For each set $S = \{a_{i_0}, a_{i_1}, ..., a_{i_t}\}$ if $t > \sqrt{n}$ then call Sort$(k \log\log n, \log_k((\log n)/4), a_{i_0}, a_{i_1}, ..., a_{i_t})$;
end

Then $(3/2)\log n$ calls to the level 1 procedure are executed. Blocks can be eliminated at most $\log n$ times. The other $(1/2)\log n$ calls are sufficient to partition the input set of size n into sets of size no larger than $\sqrt{n}$.

At level j we use only $n/k^{\log_k((\log n)/4)-j}$ words to store small integers. Each call to the Sort procedure involves a sorting on labels and a transposition of packed integers (use Lemma 6) and therefore uses linear time in terms of the number of words used. Thus the time complexity of algorithm Sort is:

$$T(level) = kT(level-1) + cn/k^{\log_k((\log n)/4)-level}; \quad (1)$$
$$T(0) = 0.$$

where c is a constant. Thus $T(\log_k((\log n)/4))$
$= O(n \log\log n / \log k)$.

We have thus proved Lemma 2.

5 Sorting in $O(n \log\log n)$ Time and Linear Space

For sorting n integers in the range $\{0, 1, 2, ..., m-1\}$ we assume that the word length used in our conservative algorithm is $O(\log(m+n))$. The same assumption is made in previous designs [2, 6, 8, 9, 14, 15]. In integer sorting we often pack several small integers into one word. We always assume that all the integers packed in a word use the same number of bits.

We take $1/\epsilon = 5$ in Andersson's exponential search tree. Thus the root has $n^{1/5}$ children and each exponential search tree rooted at a child of the root has $n^{4/5}$ leaves.

In Andersson's exponential search tree[2], integers are inserted (passed down) into the tree one at a time. Thorup[15] suggested to pass down d integers at a time, where d is the number of children of the node in the tree where integers are to be passed down. In our previous design[8, 9] we passed down d^2 integers at a time. Here we will stick with this scheme, namely passing down d^2 integers at a time. What is different from our previous design is that we will not pass down the d^2 integers all the way down the tree. Instead we will pass down one level of the tree d^2 integer at a time until all integers are passed down one level. Thus at the root we pass down $n^{2/5}$ integers at a time to the next level. After we have passed down all integers to the next level we essentially partitioned integers into $t_1 = n^{1/5}$ sets $S_1, S_2, ..., S_{t_1}$ with each S_i containing $n^{4/5}$ integers and $S_i < S_j$ if $i < j$. We then take $n^{\frac{4}{5}\cdot\frac{2}{5}}$ integers from each S_i at a time and coordinate them to be passed down to the next level of the exponential tree. We repeat this until all integers are passed down to the next level. At this time we have partitioned integers into $t_2 = n^{1/5} \cdot n^{4/25} = n^{9/25}$ sets $T_1, T_2, ..., T_{t_2}$ with each set containing $n^{16/25}$ integers and $T_i < T_j$ if $i < j$. Now we are ready

to pass integers down to the next level in the exponential search tree.

It should not be difficult to see that the tree balance operation takes $O(n \log\log n)$ time with $O(n)$ time for each level. This is the same as in the original exponential search tree proposed by Andersson[2]. For example, at the root we first take $n^{1/5}$ integers and sort them by comparison sorting. This builds one level of the exponential search tree. We then start to pass integers down the level. If the number of integers at a child exceeds $2n^{4/5}$ we split the node into two nodes. Thus at the end of this passing down we end up with at most $2n^{1/5}$ children for the root. We then regroup them to form exactly $n^{1/5}$ sets $S_1, S_2, ..., S_{t_1}$ as mentioned above.

We shall number the levels of the exponential search tree top down so that root is at level 0. Now consider the passing down at level s. Here we have $t = n^{1-(4/5)^s}$ sets $U_1, U_2, ..., U_t$ with each set containing $n^{(4/5)^s}$ integers and $U_i < U_j$ if $i < j$. Because each node at this level has $p = n^{(1/5)(4/5)^s}$ children at level $s+1$ we will pass down $q = n^{(2/5)(4/5)^s}$ integers for each set, or a total of $qt \geq n^{2/5}$ integers for all sets, at a time.

The pass down can be viewed as sorting q integers in each set together with the p integers $a_1, a_2, ..., a_p$ in the exponential search tree so that these q integers are partitioned into $p+1$ sets $S_0, S_1, ..., S_p$ such that $S_0 < \{a_1\} < S_1 < \{a_2\} < ... < \{a_p\} < S_p$.

Since we do not have to totally sort the q integers and $q = p^2$. A temptation is to use Lemma 2 to sort. For that we need nonconservative advantage which we will derive below. We will use linear timed multi-dividing technique to accomplish this.

In Section 7 of [8] it is shown that sorting the integers down the exponential search tree takes no more than $O(n\sqrt{\log\log n})$ time per level. Therefore we assume we have already sorted to level $l = 2\log\log\log n$ and we are considering the sorting down the levels greater than $2\log\log\log n$.

We use signature sorting[3] to accomplish multi-dividing. We adapt signature sorting to work for us as follows. Suppose we have a set T of p integers already sorted as $a_1, a_2, ..., a_p$ and we wish to use the integers in T to partition a set S of q integers $b_1, b_2, ..., b_q$ to $p+1$ sets $S_0, S_1, ...S_p$ such that $S_0 < \{a_1\} < S_1 < ... < \{a_p\} < S_p$. We will call this as partitioning q integers by p integers. Let $h = \log n/(c \log p)$ for a constant $c > 1$. $h/\log\log n$ $\log p$-bit integers can be stored in one word such that each word contains only $(\log n)/(c \log\log n)$ bits. We first view the bits in each a_i and each b_i as of $h/\log\log n$ segments of equal length. We view each segment as an integer. To gain nonconservative advantage for sorting we hash the integers in these words (a_i's and b_i's) to get $h/\log\log n$ hashed values in one word. In order to have intermediate values in the computing of hash values do not interfere between adjacent segments we can separate even and odd segments into two words by applying a suitable mask. We then compute hash values for the two words and then combine the hashed values

of these two words into one. Let a_i' be the hashed word corresponding to a_i and b_i' be the hashed word corresponding to b_i. Note that the hashed values total has $(2\log n)/(c\log\log n)$ bits. However, these hashed values are separated into $h/\log\log n$ segments in each words. There are "null spaces" between two adjacent segments. We can set these "null spaces" to 0's by applying a mask. We first pack all segments into $(2\log n)/(c\log\log n)$ bits(details below, the $\log\log n$ in the denominator is needed for this purpose). Now we view each hashed word as an integer and sort all these hashed words (this sorting which takes linear time will be described in detail below). After this sorting the bits in a_i and b_i are cut to $(\log\log n/h)$-th. Thus we have additional multiplicative advantage of $h/\log\log n$.

After repeating the above process g times we gain nonconservative advantage of $(h/\log\log n)^g$ while we expend only $O(gqt)$ time because each multi-dividing is done in linear ($O(qt)$) time.

The hashing function we used for hashing is obtained as follows. Because we will hash segments which are $\log\log n/h$-th, $(\log\log n/h)^2$-th,... of the whole integer, we will use hash functions for segments which are $\log\log n/h$-th, $(\log\log n/h)^2$-th.... of the whole integer. The hash function for segments which are $(\log\log n/h)^t$-th of the whole integer is obtained by cutting each of the p integers into $(h/\log\log n)^t$ segemtns. Viewing each segment as an integer we obtain $p(h/\log\log n)^t$ integers. We then obtain one hash function for these $p(h/\log\log n)^t$ integers. Because $t < \log n$ we obtain no more than $\log n$ hash functions.

Now let us take a look at the linear time sorting we mentioned earlier. Assume that we have packed the hashed values for each word into $(2\log n)/(c\log\log n)$ bits. We have t sets with each set containing $q+p$ hashed words of $(2\log n)/(c\log\log n)$ bits each. These integers are to be sorted within each set. If we sort each set individually we cannot achieve linear time. What we do is to combine all hashed words into one pool and sort them as follows.

Procedure **Linear-Time-Sort**
Input: there are $r \geq n^{2/5}$ integer d_i's. $d_i.value$ is the integer value of d_i which has $(2\log n)/(c\log\log n)$ bits. $d_i.set$ is the set d_i is in. Note that there are only t sets.
begin
1. Sort all d_i's by $d_i.value$ using bucket sort. Assume that the sorted integers are in $A[1..r]$. This step takes linear time because there are at least $n^{2/5}$ integers to be sorted.
2. **for** $j = 1$ **to** r **do**
Put $A[j]$ into set $A[j].set$;

Thus we have all sets sorted in linear time.

As we have said that after g times of reduction of bits we have nonconservative advantage $(h/\log\log n)^g$. We do not carry this bits reduction to the end because after we gained suffi-

cient nonconservative advantage we can switch to Lemma 2 for completion of partitioning q integers by the p intergers for each set. Note that by the nature of bits reduction, the original partitioning problem (partition q integers by p integers) for each set has been transformed to w partitioning subproblems on w subsets, for some integer w (See section 2 and Fig. 1 for explanation).

Now for each set we combine all its subsets in subproblems into one set. We then invoke Lemma 2 to do the partition. Because we have $(h/\log\log n)^g$ nonconservative advantage the algorithm in Lemma 2 takes $O(qt\log\log n/(g(\log h - \log\log\log n) - \log\log\log n))$ time. Let $qtg = qt\log\log n\ /(g(\log h - \log\log\log n) - \log\log\log n)$. We arrive at $g = (\log\log n/(\log h - \log\log\log n - (\log\log\log n)/g))^{1/2} < (\log\log n/(\log h - 2\log\log\log n))^{1/2}$. Since we have assumed that we are working on levels greater than $2\log\log\log n$ we need to sum g for $\log h = 2\log\log\log n$ to $\log\log n$, for which we have $\sum_{\log h=2\log\log\log n}^{\log\log n} g \leq \sum_{\log h=2\log\log\log n}^{\log\log n}(\log\log n/(\log h - 2\log\log\log n))^{1/2} = O(\log\log n)$.

We have partitioned q integers by p integers in each set. Thus we have $S_0 < \{e_1\} < S_1 < ... < \{e_p\} < S_p$, where e_i is a segment of a_i obtained by bits reduction. (Because bits reduction each of the p integers could produce several segments and therefore we could have more than p but less than $p\log n$ e_i's and S_i's. But this does not affect the analysis of our algorithm.) What we have done the partitioning is by combining all subsets of subproblems. Assume integers are stored in array B such that integers in S_i precede integers in S_j if $i < j$. And e_i is stored after S_{i-1} and before S_i. Let $B[i]$ in subset $B[i].subset$. To let the partitioning be done for each subset we do the following:

for $j = 1$ **to** q **do**
Put $B[j]$ into subset $B[j].subset$.

This takes linear time and $O(n)$ space.

Now we are back to the packing problem which we solve as follows. We can assume that the number of bits $\log m$ in a word satisfying $\log m \geq \log n\log\log n$, for otherwise we can use radix sort to sort the integers. A word has $h/\log\log n$ hashed values (segments) in it at level $\log h$ of the exponential search tree. The total number of hashed bits in a word is $(2\log n)/(c\log\log n)$ bits. Therefore the hashed bits in a word looks like $0^i t_1 0^i t_2 0^i ... t_{h/\log\log n}$, where t_k's are hashed bits and 0^i are the null spaces between hashed bits. We first pack $\log\log n$ words into one word to get $w_1 = 0^j t_{11} t_{21}\ ...\ t_{\log\log n,1} 0^j t_{12} t_{22}\ ...\ t_{\log\log n,2} 0^j\ ...\ t_{1,h/\log\log n} t_{2,h/\log\log n}\ ...\ t_{\log\log n,h/\log\log n}$, where $t_{i,k}$'s, $k = 1, 2, ..., h/\log\log n$, are from the i-th word. We then use $O(\log\log n)$ steps to pack w_1 to $w_2 = 0^{jh/\log\log n} t_{11} t_{21}\ ...\ t_{\log\log n,1} t_{12} t_{22}\ ...\ t_{\log\log n,2}\ ...\ t_{1,h/\log\log n} t_{2,h/\log\log n}\ ...\ t_{\log\log n,h/\log\log n}$. Now the packed hash

bits in w_2 has only $2\log n/c$ bits. We use $O(\log\log n)$ time to unpack w_2 to $\log\log n$ words $w_{3,k} = 0^{jh/\log\log n}0^r t_{k1}0^r\ t_{k2}0^r ... t_{k,h/\log\log n}$, $k = 1, 2, ..., \log\log n$. We then use $O(\log\log n)$ time to pack these $\log\log n$ words into one word $w_4 = 0^r t_{11}0^r t_{12}0^r t_{13}0^r\ ...\ t_{1,h/\log\log n}0^r t_{21}0^r t_{22}0^r\ ...\ t_{2,h/\log\log n}0^r\ ...\ t_{\log\log n,1}0^r t_{\log\log n,2}0^r\ ...\ t_{\log\log n,h/\log\log n}$. We then use $O(\log\log n)$ steps to pack w_4 to $w_5 = 0^s t_{11}t_{12}t_{13}\ ...\ t_{1,h/\log\log n}t_{21}t_{22}\ ...\ t_{2,h/\log\log n}\ ...\ t_{\log\log n,1}t_{\log\log n,2}\ ...\ t_{\log\log n,h/\log\log n}$. We finally use $O(\log\log n)$ steps to unpack w_5 to $\log\log n$ packed words. Overall we expended $O(\log\log n)$ time for packing $\log\log n$ words. Thus for each word the time expended is constant.

Thus we have:

Theorem 1: n integers can be sorted in $O(n\log\log n)$ time and linear space.

6 Conclusions

We have finally achieved $O(n\log\log n)$ time and linear space for integer sorting. Although it is not known whether this is the lower bound, we believe that breaking this bound deterministically should be very difficult.

References

[1] S. Albers and T. Hagerup. Improved parallel integer sorting without concurrent writing. *Information and Computation*, **136**, 25-51(1997).

[2] A. Andersson. Fast deterministic sorting and searching in linear space. *Proc. 1996 IEEE Symp. on Foundations of Computer Science*, 135-141(1996).

[3] A. Andersson, T. Hagerup, S. Nilsson, R. Raman. Sorting in linear time? *Proc. 1995 Symposium on Theory of Computing*, 427-436(1995).

[4] P. van Emde Boas, R. Kaas, E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Syst. Theory* **10** 99-127(1977).

[5] M. Dietzfelbinger, T. Hagerup, J. Katajainen, M. Penttonen. A reliable randomized algorithm for the closest-pair problem. *J. Algorithms* **25**, 19-51(1997).

[6] M.L. Fredman, D.E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. System Sci.* 47, 424-436(1994).

[7] T. Hagerup and H. Shen. Improved nonconservative sequential and parallel integer sorting. *Infom. Process. Lett.* **36**, pp. 57-63(1990).

[8] Y. Han. Improved fast integer sorting in linear space. *Information and Computation*, Vol. 170, No. 1, 81-94(Oct. 2001).

[9] Y. Han, Fast integer sorting in linear space. *Proc. Symp. Theoretical Aspects of Computing (STACS'2000), Lecture Notes in Computer Science 1170*, 242-253(Feb. 2000).

[10] Y. Han, M. Thorup. Sorting integers in $O(n\sqrt{\log\log n})$ expected time and linear space. Manuscript.

[11] 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(August, 1995).

[12] D. Kirkpatrick and S. Reisch. Upper bounds for sorting integers on random access machines. *Theoretical Computer Science* **28**, 263-276(1984).

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

[14] R. Raman. Priority queues: small, monotone and trans-dichotomous. *Proc. 1996 European Symp. on Algorithms, Lecture Notes in Computer Science 1136*, 121-137(1996).

[15] M. Thorup. Fast deterministic sorting and priority queues in linear space. *Proc. 1998 ACM-SIAM Symp. on Discrete Algorithms (SODA'98)*, 550-555(1998).

[16] M. Thorup. Randomized sorting in $O(n \log\log n)$ time and linear space using addition, shift, and bit-wise boolean operations. *Proc. 8th ACM-SIAM Symp. on Discrete Algorithms (SODA'97)*, 352-359(1997).