Improved Fast Integer Sorting in Linear Space[1]

Yijie Han

Computer Science Telecommunications Program
University of Missouri — Kansas City
5100 Rockhill Road
Kansas City, MO 64110, USA
han@cstp.umkc.edu
http://welcome.to/yijiehan

[1]Preliminary versions of this paper have been presented at 2000 Symposium on Theoretical Aspects of Computing(STACS'2000) and at 2001 ACM-SIAM Symposium on Discrete Algorithms (SODA'2001).

Running head: Integer sorting

Mailing address:

Yijie Han
Computer Science Telecommunications Program
University of Missouri - Kansas City
5100 Rockhill Road
Kansas City, MO 64110

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 $O(n \log\log n \log\log\log n)$ time. When $\log m \geq \log^{2+\epsilon} n$, $\epsilon > 0$, we can further achieve $O(n \log\log n)$ time. This improves the $O(n(\log\log n)^2)$ time bound given in Thorup(1998). This result is obtained by combining our new technique with that of Thorup's (1998). Signature sorting (Andersson, Hagerup, Nilsson, Raman 1995), Anderson's result (Andersson 1996), Raman's result (Raman 1996), our previous result (Han and Shen 1999) are also used for the design of our algorithms. We provide an approach and techniques which are totally different from previous approaches and techniques for the problem. As a consequence our technique can be extended to apply to nonconservative sorting and parallel sorting. Our nonconservative sorting algorithm sorts n integers in $\{0, 1, ..., m-1\}$ in time $O(n(\log\log n)^2/(\log k + \log\log\log n))$ using word length $k \log(m+n)$, where $k \leq \log n$. Our EREW parallel algorithm sorts n integers in $\{0, 1, ..., m-1\}$ in $O((\log n)^2)$ time and $O(n(\log\log n)^2/\log\log\log n)$ operations provided $\log m = \Omega((\log n)^2)$.

Keywords: Algorithms, bucket sorting, integer sorting, linear space, parallel sorting.

List of symbols: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, a, b, c, d, e, f, g, h, I, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, (,), @, *, =, [,], {, }, Ω, ϵ, $\square$.

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 the sequential and parallel integer sorting (Albers and Hagerup 1997; Andersson 1996; Andersson, Hagerup, Nilsson, Raman 1995; Bhatt, Diks, Hagerup, Prasad, Radzik, Saxena 1991; Dessmark and Lingas 1998; Fredman and Willard 1994; Hagerup 1987; Hagerup and Shen 1990; Han and Shen 1995, 1999; Kirkpatrick and Reisch 1984; Kruskal, Rudolph, Snir 1990; Rajasekaran and Reif 1989; Rajasekaran and Sen 1992; Raman 1996; Thorup 1997, 1998; Vaidyanathan, Hartman, Varshney 1993; Wagner and Han 1986). Recent advances in the design of algorithms for integers sorting have resulted in fast algorithms (Andersson, Hagerup, Nilsson, Raman 1995; Han and Shen 1995; Thorup 1997). However, these algorithms use randomization or superlinear space. For sorting integers in $\{0, 1, ..., m-1\}$ $O(nm^{\epsilon})$ space is used in the algorithms reported in (Andersson, Hagerup, Nilsson, Raman 1995; Han and Shen 1995), 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(1994) shows that n integers can be sorted in $O(n \log n / \log\log n)$ time in linear space. Raman (1996) showed that sorting can be done in $O(n\sqrt{\log n \log\log n})$ time in linear space. Later Andersson (1996) improved the time bound to $O(n\sqrt{\log n})$. Then Thorup (1998) improved the time bound to $O(n(\log\log n)^2)$. In this paper we further improve upon previous results. We show that n integers in $\{0, 1, 2, ..., m-1\}$ can be sorted in $O(n \log\log n \log\log\log n)$ time in linear space. When $\log m > \log^{2+\epsilon} n$, $\epsilon > 0$, we further achieve $O(n \log\log n)$ time.

Our approach and technique for the design of integer sorting algorithm are of independent interest. They differ totally from previous approaches and techniques. The approach used by Fredman and Willard(1994), by Raman(1996), by Andersson(1996), and by Thorup(1998) is the design of search trees or priority queues to support insert, delete, update and other operations. By using an exponential search tree Andersson was able to achieve the $O(n\sqrt{\log n})$ time complexity. Thorup(1998) also uses search trees and priority queues to achieve the $O(n(\log\log n)^2)$ time bound. Our approach to the integer sorting problem is to relate the subdivision of integers and the number of bits sorted. A new technique is presented which converts sorting on large integers to the sorting on very small integers. The optimality of the well known bucket sorting algorithm sorting n integers in $\{0, 1, 2, ..., n-1\}$ in $O(n)$ time and space then enables us to speed up sorting for arbitrarily large integers. In summary our approach and technique show that arbitrarily large integers can be sorted by merely sorting on very small integers.

Our approach and technique alone will yield a linear space sorting algorithm with time complexity $O(n(\log\log n)^2/\log\log\log n)$. Although this improves on the best previous result the improvement is not large. By combining our algorithm with that of Thorup's and by applying signature sorting (Andersson, Hagerup, Nilsson, Raman 1995), Andersson's result (Andersson 1996), Raman's result (Raman 1996) and our previous result (Han and Shen 1999) we show that time complexity $O(n\log\log n\log\log\log n)$ can be achieved for sorting n integers in $\{0,1,2,...,m-1\}$ in linear space. When $\log m > \log^{2+\epsilon} n$, $\epsilon > 0$, we can further achieve $O(n\log\log n)$ time.

Our techniques are based on a kind of vector grouping: as in many previous approaches, we are viewing a word as a vector of k characters (also referred to as small integers). Together with each such vector, we have vector of labels. The i-th label is the label of the i-th character. We will have n/k such vector pairs, and the goal is now to permute all the characters in all the words so that characters with the same label come together. We will show (quite simply) that if k labels take up only $\log n/2$ bits, we can do the grouping in $O((n/k)\log k)$ time.

To appreciate the strength of the above result, note that one of the basic results in the area of Albers and Hagerup (1997) is that we can sort the characters, distributed over n/k words, in $O(n(\log n)(\log k)/k)$ time. This would easily give us the grouping, but be slower by a factor $\log k\log n/\log\log n$.

The efficiency of the above vector grouping allows us to efficiently sort our full-word integers, one block at a time, starting with the most significant block. The labels will keep track of the sorting done of more significant characters when dealing with less significant characters.

We would like to point out that previous approaches(Andersson 1996; Fredman and Willard 1994; Raman 1996) solve the integer sorting problem by repeated operation of inserting **single** integer into the search tree. Thorup(1998) finds that by inserting integers in batches the time bound can be improved. Our approach and technique also show that advantage can be taken when integers are sorted in groups, especially when this group is large (say containing close to n integers). Milterson(1994) has shown that dynamic searching takes $\Omega((\log n)^{1/3})$ time. Therefore sorting integers in groups does provide provable advantage. Note that in contrast repeated inserting single element into a search tree can result in an optimal algorithm for comparison sorting.

Unlike previous techniques (Andersson 1996; Fredman and Willard 1994; Raman 1996; Thorup 1998), our technique can be extended to apply to nonconservative sorting and parallel sorting. Conservative sorting is to sort n integers in $\{0,1,...m-1\}$ with word length (the number of bits in a word) $O(\log(m+n))$ (Kirkpatrick and Reisch 1984). Nonconservative sorting is to sort with word length larger than $O(\log(m+n))$. We show that n integers in $\{0,1,...,m-1\}$ can be sorted in time $O(n(\log\log n)^2/(\log k+\log\log\log n))$ with word length $k\log(m+n)$ where $k \leq \log n$.

Thus if $k = (\log n)^\epsilon$, $0 < \epsilon < 1$, the sorting can be done in linear space and $O(n \log\log n)$ time. Andersson(1996) and Thorup(1998) did not show how to extend their linear space sorting algorithm to nonconservative sorting. Thorup(1998) used an algorithm to insert a batch of n integers into the search tree in $O(n \log\log n)$ time. When using word length $k \log(m+n)$ this time complexity can be reduced to $O(n(\log\log n - \log k))$, thus yielding an $O(n \log\log n(\log\log n - \log k))$ time algorithm for linear space sorting, which is considerably worse than our algorithm.

Also note that previous results (Andersson 1996; Fredman and Willard 1994; Thorup 1998) do not readily extend to parallel sorting. Our technique can be applied to obtain a more efficient parallel algorithm for integer sorting. In this regard the best previous result on the EREW PRAM is due to Han and Shen(1999) which sorts n integers in $\{0, 1, ..., m-1\}$ in $O(\log n)$ time and $O(n\sqrt{\log n})$ operations(time processor product). We show when $\log m = \Omega((\log n)^2)$ we can sort in $O((\log n)^2)$ time with $O(n(\log\log n)^2/\log\log\log n)$ operations on the EREW PRAM. Thus for large integers our new algorithm is more efficient than the best previous algorithm.

2 Preparation

Our algorithm is built upon Andersson's exponential search tree (Andersson 1996). 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 (Andersson 1996) 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. Anderson 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 (Thorup 1998) 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.

We speed up the integer sorting by using a batch of size d^2 and by using nonconservative sorting (defined below). In Section 7 (also Han 2000) we first show that we can insert integers from one level of the tree to the next level in amortized $O(\sqrt{\log\log n})$ time resulting in an $O(n(\log\log n)^{3/2})$ time algorithm for linear space sorting. We then speed up this insertion process further in Section 8 by resorting to multi-dividing, i.e. cutting the number of bits in an integer into nonconstant number of segments and discarding all segments but one. Multi-dividing is accomplished by signature sorting (Andersson, Hagerup, Nilsson, Raman 1995). We first apply multi-dividing to the integers in the buffer at each node of the tree. When the size of the integer is sufficiently small we then apply

nonconservative sorting to insert the integers to the next level.

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.

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 Emde Boas et al. (P. van Emde Boas, R. Kaas, E. Zijlstra 1977). 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 (Andersson 1996) 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.* (M. Dietzfelbinger, T. Hagerup, J. Katajainen, M. Penttonen 1997) and Raman (Raman 1996) 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 Raman 1996): 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 has 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.* (A. Andersson, T. Hagerup, S. Nilsson, R. Raman 1995) 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 (Andersson 1996, Raman 1996). Obviously Raman's hash function does not satisfy this criterion when b is large. However, Andersson's result (Andersson 1996) 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.

In addition to utilizing the above results we also make use of Andersson (Andersson 1996) and Thorup's construction (Thorup 1998). In (Andersson 1996) Andersson builds an exponential search tree. Thorup uses this exponential search tree and associates buffer $B(v)$ with each node v of the tree. He defines that a buffer $B(v)$ is over-full if $|B(v)| > d(v)$, where $d(v)$ is the number of children of v. When the buffer is over-full the integers of the buffer are flushed to the next lower level of the exponential search tree. In Thorup's algorithm the flush of buffer $B(v)$ takes $O(|B(v)| \log\log |B(v)|)$ time, where $|B(v)|$ is the number of integers in $B(v)$.

3 Sorting on Small Integers

Word length is the number of bits in a word. 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 (Andersson 1996; Fredman and Willard 1994; Raman 1996; Thorup 1998). 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. Suppose k integers each having l bits are packed into one word. By using the test bit technique (Albers and Hagerup 1997; Andersson, Hagerup, Nilsson, Raman 1995) 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 2: Selecting the s-th largest integer among the n integers packed into n/k words can be done in $O(n \log k/k)$ time and $O(n/k)$ space. In particular the median can be found in $O(n \log k/k)$ time and $O(n/k)$ space.

Proof: Since we can do pairwise comparison of k integers in one word with k 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, ... k-th integer of 5 words into one word in constant time. Thus the set S of medians are now contained in $n/(5k)$ 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/k words into $3n/(4k)$ words (the packing incurs the factor $\log k$ in the time complexity) and then recurse. Packing can be done by the packing algorithm in (Leighton 1992) (Section 3.4.3). □

Now consider sorting small integers. Let k integers be packed in one word. We say that the nk integers in n words are sorted if ki-th to $(k(i+1)-1)$-th smallest integers are sorted and packed in the i-th word, $0 \leq i < n$. We have the following lemma:

Lemma 3: If k integers using a total of $(\log n)/2$ bits are packed into one word, then the n integers in n/k words can be sorted in $O((n/k) \log k)$ time and $O(n/k)$ space.

Proof: Because only $(\log n)/2$ bits are used in each word to store k integers we can use bucket sorting to sort all words by treating each word as one integer and this takes $O(n/k)$ 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 $k < (\log n)/2$ words with the same pattern into one group. For each pattern there are at most $k-1$ words left which cannot form a group. Therefore at most $\sqrt{n} \cdot (k-1)$ words cannot form groups. For each group we move the i-th integer in all k words into one word. That is, we take k k-integer vectors and produce k k-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 (1997) in time $O(k \log k)$ and space $O(k)$. Therefore for all groups it takes $O((n/k) \log k)$ time and $O(n/k)$ space.

For the words not in a group (there are at most $\sqrt{n} \cdot (k-1)$ of them) we simply disassemble the words and then reassemble the words. This will take no more than $O(n/k)$ 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 $k = O(\log n)$ we are sorting $O(n)$ integers packed in n/k words in $O((n/k) \log \log n)$ time and $O(n/k)$ space. Therefore the saving is considerable.

Lemma 4: Assume that each word has $\log m > \log n$ bits, that k integers each having $(\log m)/k$

bits are packed into one word, that each integer has a label containing $(\log n)/(2k)$ bits, and that the k 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/k words can be sorted *by their labels* in $O((n \log\log n)/k)$ time and $O(n/k)$ 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 3. We can transpose each group of words for integers. □

Note also that the sorting algorithm given in Lemma 3 and Lemma 4 are not stable. As will be seen later we will use these algorithms to sort arbitrarily large integers. Even though we do not know how to make the algorithm in Lemma 3 stable, as will be seen that our sorting algorithm for sorting large integers 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 5: Assume that each word has $\log m \log\log n > \log n$ bits, that k integers each having $(\log m)/k$ bits are packed into one word, that each integer has a label containing $(\log n)/(2k)$ bits, and that the k labels are packed into one word the same way as integers are packed into words, then n integers in n/k words can be sorted by their labels in $O(n/k)$ time and $O(n/k)$ 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 3 and 4 we sort the words containing packed labels by bucket sorting. In order to transpose words of integers we put $k \log\log n$ words of integers into one group instead of putting k words of integers into one group. To transpose the integers in a group containing $k \log\log n$ words we first further pack $k \log\log n$ words into k words by packing $\log\log n$ words of integers into one word. We then do transpose on the k words. Thus transpose takes only $O(k \log\log n)$ time for each group and $O(n/k)$ time for all integers. After finishing transpose we then unpack the integers in the k words into $k \log\log n$ words. □

Note also if the word length is $\log m \log\log n$ and only $\log m$ bits are used to pack $k \leq \log n$ integers into one word. Then the selection in Lemma 2 can be done in $O(n/k)$ time and space because the packing in the proof of Lemma 2 can now be done in $O(n/k)$ time.

4 The Approach and The Technique

Consider the problem of sorting n integers in $\{0, 1, ..., m-1\}$. We assume that each word has $\log m$ bits and 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 sort from high order bits to low order bits. We now propose 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 2 we can find the median of these n small integers in $O(n \log \log n / \log n)$ time 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 $O(n \log \log n)$ 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 4. Since the total label size in each word has to be $\log n/2$, and each label uses $\log e$ bits, the number k of labels in each word has to be at most $\log n/(2 \log e)$. Further, since $k = \log n/(2 \log e)$ small integers should fit in a word, and each word contains $\log n$ blocks, each small integer can contain $O(\log n/k) = O(\log e)$ blocks. Thus we assume that $(\log n)/(2 \log e)$ small integers each containing $\log e$ 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 4 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 $O((n\log\log n\log e)/\log n)$.

Note that $O(\log e)$ blocks for each small integer is the most number of bits we can move in applying Lemma 4 because each word has $\log m$ bits. Note also that the moving process is not stable as the sorting algorithm in Lemma 4 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 proceed 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 $\log e$ continuous blocks is the current block. Thus after we move these $\log e$ blocks into the set we delete the original current block because we know that the $\log e$ 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 $\log e$ blocks.

Another problem we have to pay attention to is that the size of the sets after several stages of dividing will become small. The scheme of Lemmas 3 and 4 relies on the fact that the size of the set is not very small. We cope with this problem in this way. If the size of the set is larger than $\sqrt{n}$ we keep dividing the set. In this case each word for packing the labels can use at least $(\log n)/4$ bits. When the size of the set is no larger than $\sqrt{n}$ we then use a recursion to sort the set. In each next level of recursion each word for packing the labels uses less number of bits. The recursion has

$O(\log\log n)$ levels.

Below is our sorting algorithm which is used to sort integers into sets of size no larger than $\sqrt{n}$. This algorithm uses yet another recursion (do not confuse this recursion with the recursion mentioned in the above paragraph).

Algorithm Sort$(level, a_0, a_1, ..., a_t)$

/* 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 2 to find the median and then using Lemma 4 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. Cut the bits in each integer a_i into equal two segments a_i^{High} (high order bits) and a_i^{Low} (low order bits). Pack a_i^{High}'s into half the number of words. Call Sort$(level-1, a_0^{High}, a_1^{High}, ..., a_t^{High})$. /*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. */

3. For each integer a_i extract out a_i^{Low} which has half the number of bits as in a_i and is a continuous segment with the most significant block being the current block of a_i. Pack a_i^{Low}'s into half the number of words as in the input. Route a_i^{Low}'s to their sets by using Lemma 4.

4. For each set $S = \{a_{i_0}, a_{i_1}, ..., a_{i_s}\}$ call Sort$(level-1, a_{i_0}^{Low}, a_{i_1}^{Low}, ..., a_{i_s}^{Low})$.

5. Route the label which is the set number for each integers back to the position where the integer located in the input to the procedure call. Also route a number (a $2(level+1)$ bit number) for each integer indicating the current block back to the location of the integer. This step is the reverse of the routing in Step 3.

In Step 3 of algorithm Sort we need to extract a_i^{Low}'s and to pack them. The extraction requires a mask. This mask can be computed in $O(\log\log n)$ time for each word. Suppose k small integers each containing $(\log n)/(4k)$ blocks are packed in a word. We start with a constant which is $(0^{(t\log n)/(8k)}1^{(t\log n)/(8k)})^k$ when represented as a binary string, where t is the number of bits in a

block. Because a $2(level+1)$ bit number a is used to note the current block we can check 1 bit of a in a step for all a's packed in a word (there are k of them). This can determine whether we need to shift the $1^{(t\log n)/(8k)}$ for each small integer to the left or not. Thus using $O(\log\log n)$ time we can produce the mask for each word. Suppose the current block is the $((\log n)/(8k)+g)$-th block then the resulting mask corresponding to this small integer will be $0^{t((\log n)/(8k)-g)}1^{(t\log n)/(8k)}0^{tg}$. Packing is to pack $s \leq \log n$ blocks to consecutive locations in a word. This can be done in $O(\log\log n)$ time for each word by using the packing algorithm in (Leighton 1992)(Section 3.4.3).

We let a block contain $(4\log m)/\log n$ bits. Then if we call Sort$(\log((\log n)/4), a_0, a_1, ..., a_{n-1})$ 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. When the procedure returns the number of eliminated bits in different sets could be different. Therefore we need modify our procedure a little bit. At level j we form a_i^{High} by extract out the 2^{j-1} continuous blocks with the most significant block being the current block from a_i. After this modification we call Sort several times as below:

Algorithm IterateSort
Call Sort$(\log((\log n)/4), a_0, a_1, ..., a_{n-1})$;
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$(\log((\log n)/4), a_{i_0}, a_{i_1}, ..., a_{i_t})$;
  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 $n/2^{\log((\log n)/4)-j}$ words to store small integers. Each call to the Sort procedure involves a sorting on labels and a transposition of packed integers (use Lemma 4) and therefore involves a factor of $\log\log n$ in time complexity. Thus the time complexity of algorithm Sort is:

$$T(level) = 2T(level-1) + cn\log\log n/2^{\log((\log n)/4)-level}; \quad (1)$$
$$T(0) = 0.$$

where c is a constant. Thus $T(\log((\log n)/4)) = O(n(\log\log n)^2)$. Algorithm IterateSort only sorts sets into sizes less than $\sqrt{n}$. We need another recursion to sort sets of size less than $\sqrt{n}$.

This recursion has $O(\log\log n)$ levels. Thus the time complexity to have the input integers sorted is $O(n(\log\log n)^3)$.

The sorting process is not stable. Since we are sorting arbitrarily large integers we can append the address bits to each input integer to stabilize the sorting. Although this requires that each word contains $\log m + \log n$ bits, when $m \geq n$ the number of bits for each word can be kept at $\log m$ by using the idea of radix sorting, namely sorting $\frac{\log m}{2} + \log n$ bits in each pass.

The space used for each next level of recursion in Sort uses half the size of the space. After recursion returns the space can be reclaimed. Thus the space used is linear, i.e. $O(n)$.

Theorem 1: n integers can be sorted in linear space in time $O(n(\log\log n)^3)$. □

5 An Algorithm with Time Complexity $O(n(\log\log n)^2)$

We first note the following Lemma.

Lemma 6: If the word length used in the algorithm is $\log m \log\log n$. then n integers in $\{0, 1, ..., m-1\}$ can be sorted into sets of size no larger than $\sqrt{n}$ in linear space in time $O(n\log\log n)$.

Proof: In this case the median finding takes linear time and we can use Lemma 5 to sort packed small integers. Also it takes $O(\log\log n)$ time to extract out a_i^{High}'s and a_i^{Low}'s for $\log\log n$ words (including computing mask and packing) because we can pack $\log\log n$ words further into one word. Therefore formula (1) becomes:

$$T(level) = 2T(level-1) + cn/2^{\log((\log n)/4)-level}; \quad (2)$$
$$T(0) = 0.$$

Therefore $T(\log((\log n)/4)) = O(n\log\log n)$. That is, the time complexity for dividing the input set to sets of size no larger than $\sqrt{n}$ is $O(n\log\log n)$. □

We apply the following technique to improve the time complexity of our algorithm further.

We divide $\log m$ bits of an integers into $\log(setsize)\log\log n$ blocks with each block containing $(\log m)/(\log(setsize)\log\log n)$ bits, where $setsize$ is the size of the set we are to sort into. Initially $setsize = \sqrt{n}$. We execute the following algorithm with $setsize = \sqrt{n}$:

Algorithm SpeedSort($setsize$)
while there is a set S which has size $> setsize$ **do**
begin

1. for each integer $a_i \in S$ extract out a_i' which contains $\log setsize$ continuous blocks of a_i with the most significant block being the current block, put all a_i''s in S';
2. Call IterateSort on set S';

end

The whole sorting process consists of $\log\log n$ levels of calling SpeedSort: SpeedSort($n^{1/2}$), SpeedSort($n^{1/4}$), SpeedSort($n^{1/8}$), ... SpeedSort($n^{1/2^i}$), ... In SpeedSort($n^{1/2^i}$) each word stores $\log(setsize) = \log n/2^i$ blocks and each block contains $\log m/(\log(setsize)\log\log n) =$ $2^i \log m/(\log n \log\log n)$ bits. Therefore during the sorting process each word stores no more than $\log m/\log\log n$ bits of integer data. By Lemma 6 one iteration of the while loop in any of the SpeedSort's takes $O(\log\log n)$ time for each integer. We account the time for each integer in the whole sorting process by two variables D and E. If an integer a has gone through g_i iterations of the while loop of SpeedSort($n^{1/2^i}$) then $(g_i - 1)\log m/\log\log n$ bits of a has been eliminated in SpeedSort($n^{1/2^i}$). We add $O((g_i - 1)\log\log n)$ to variable E indicating that that much time has been expended to eliminate $(g_i - 1)\log m/\log\log n$ bis. We also add $O(\log\log n)$ time to variable D indicating that that much time has been expended to divide the set in SpeedSort($n^{1/2^i}$). Because we can eliminate at most $\log m$ bis therefore the value of E is upbounded by $\sum_i (g_i \log\log n) =$ $O((\log\log n)^2)$ throughout all levels of SpeedSort invocations. The value of variable D is also upbounded by $O((\log\log n)^2)$ because there are $\log\log n$ levels of SpeedSort invocations. Therefore we have

Theorem 2: n integers can be sorted in linear space in $O(n(\log\log n)^2)$ time. □

6 Nonconservative Sorting and Parallel Sorting

When the word length is $k\log(m+n)$ for sorting integers in $\{0, 1, ..., m-1\}$ we modify algorithm Sort in Section 3. Here whenever we sort t bits in the integer we can move tk bits in step 3 of Sort. Thus in step 2 of Sort we can divide a_i into equal k segments. Subsequently we can invoke recursion k times. Each time we sort on a segment. Immediately upon finishing we move a_i to its sorted set. We can move the whole a_i instead of a segment of a_i because we have the advantage of the nonconservatism. When $k = 1$ Sort is called as in Section 3 with level numbers as follows:

$\log((\log n)/4)$, ..., 3, 2, 1, 2, 1, 3, 2, 1, 2, 1, 4, 3, 2, 1, 2, 1, 3, 2, 1, ...

The total number of different levels is $\log\log n$ which is accounted in the time complexity. When $k > 1$ the nonconservative version of Sort is called with level numbers as follows:

$\log((\log n)/4)$, ..., $3\log k$, $2\log k$, $\log k$, 1, 1, 1, ..., (total k 1's), $\log k$, 1, 1, 1, ..., (total k 1's), $\log k$, ...

The total number of different levels is $\log\log n/\log k$ which is then accounted in the time complexity. Therefore algorithm Sort can be done in $O(n(\log\log n)^2/\log k)$ time if each integer has only $O(\log m/k)$ bits. Here we assume that transposition is done in $O(n\log\log n)$ time for n words. If

we apply the technique in Section 5 we can assume that the transposition can be done in $O(n)$ time for n words. Therefore the time complexity for algorithm Sort becomes $O(n \log\log n / \log k)$. Since there are $O(\log\log n)$ calls to Sort which are made in the whole sorting process, the time complexity of our nonconservative sorting algorithm to sort n integers is $O(n(\log\log n)^2 / \log k)$.

Theorem 3: n integers in $\{0, 1, ..., m-1\}$ can be sorted in $O(n(\log\log n)^2 / \log k)$ time and linear space with word length $k \log(m+n)$ (or with nonconservative advantage k), where $1 \leq k \leq \log n$.

We can reduce the complexity in Theorem 3. We divide the process in algorithm Sort into $\log\log n/(2 \log\log\log n)$ phases. j-th phase is to divide sets of size $2^{\log n(1-\frac{j \log\log\log n}{\log\log n})}$ to sets of size $2^{\log n(1-\frac{(j+1) \log\log\log n}{\log\log n})}$. In each phase only $\frac{\log n \log\log\log n}{\log\log n}$ blocks are used for sorting. Thus each phase takes $O(n \log\log n \log\log\log n)$ time (a total of $O(n(\log\log n)^2)$ time for $\log\log n/(2 \log\log\log n)$ phases) if nonconservative advantage is 1. Now we make the similar observation as we have made in Section 5. Because there are a total of $c(\log\log n)^2 / \log\log\log n$ phases in the whole sorting process (including Sort and recursions after Sort), where c is a constant, we can divide $\log m$ bits of input integers into $c(\log\log n)^2 / \log\log\log n$ segments and use one segment at a time. Following the analysis in Section 5 we have now nonconservative advantage $c(\log\log n)^2 / \log\log\log n$. Within this nonconservative advantage we use a factor of $\log\log n$ advantage to reduce the time complexity of sorting by a factor of $\log\log n$ as in Lemma 6. The other factor of $\log\log n / \log\log\log n$ nonconservative advantage can be used as in Theorem 3 to speed up the algorithm by reducing the time complexity by a factor of $\log\log\log n$. Thus we have:

Theorem 4: n integers can be sorted by a conservative algorithm in time $O(n(\log\log n)^2 / \log\log\log n)$ and linear space. □

Theorem 5: n integers in $\{0, 1, ..., m-1\}$ can be sorted in $O(n(\log\log n)^2 / (\log\log\log n + \log k))$ time and linear space with word length $k \log(m+n)$, where $1 \leq k \leq \log n$. □

Concerning parallel integer sorting we note that on the EREW PRAM we can have the following lemma to replace Lemma 2.

Lemma 7: An integer a among the n integers packed into n/k words can be computed on the EREW PRAM in $O(\log n)$ time and $O(n/k)$ operations using $O(n/k)$ space such that a is ranked at least $n/4$ and at most $3n/4$ among the n integers.

The proof of Lemma 7 can be obtained by applying Cole's parallel selection algorithm (Cole 1987/88). Following Cole's algorithm we can keep selecting from small sets of words. We only do component-wise sorting among the integers in a set of words. That is the 1st integers in all words in a set are sorted, the 2nd integers in all words in a set are sorted, and so on. This component-wise sorting can be done using AKS sorting network (Ajtai, Komlós, Szemerédi 1983). Each node of the sorting network do pairwise comparison of the integers packed in two words. Note that we can do

without further packing integers into fewer words and therefore the factor $\log k$ does not show up in the time complexity.

Currently Lemma 3 cannot be parallelized satisfactorily. On the EREW PRAM the currently best result (Han and Shen 1999) sorts in $O(\log n)$ time and $O(n\sqrt{\log n})$ operations. To replace Lemma 3 for parallel sorting we resort to nonconservatism.

Lemma 8: If $k = 2^t$ integers using a total of $(\log n)/2$ bits are packed into one word, then the n integers in n/k words can be sorted in $O(\log n)$ time and $O(n/k)$ operations on the EREW PRAM using $O(n/k)$ space, provided that the word length is $\Omega((\log n)^2)$.

The sorting of words in Lemma 8 is done with the nonconservative sorting algorithm in (Han and Shen 1999). The transposition can also be done in $O(n)$ operations because of nonconservatism.

For Lemma 4 we have to assume that $\log m = \Omega((\log n)^2)$. Then we can sort the n integers in n/k words by their labels in $O(\log n)$ time and $O((n \log\log n)/k)$ operations on the EREW PRAM using $O(n/k)$ space. Note here that labels are themselves being sorted by nonconservative sorting algorithm in Lemma 8. Note also that the transposition of integers packed in words here incurs a factor of $\log\log n$ in the operation complexity.

Lemma 5 and Section 5 say how do we remove the factor $\log\log n$ from the time complexity incurred in transposition with nonconservatism. This applies to parallel sorting as well to reduce the factor of $\log\log n$ from the operation complexity.

Because algorithm Sort uses algorithms in Lemmas 2 to 4 $O(\log n)$ times and because we can now replace Lemmas 2 to 4 with corresponding Lemmas for parallel computation, algorithm Sort is in effect converted into a parallel EREW PRAM algorithm with time complexity $O((\log n)^2)$ and operation complexity $O(n(\log\log n)^2)$. The techniques in Section 5 and in Theorem 4 apply to parallel sorting. Therefore we have

Theorem 6: n integers in $\{0, 1, ..., m-1\}$ can be sorted in $O((\log n)^2)$ time and $O(n(\log\log n)^2 / \log\log\log n)$ operations provided that $\log m = \Omega((\log n)^2)$.

Note that although algorithm Sort takes $O((\log n)^2)$ time, the whole sorting algorithm takes $O((\log n)^2)$ time as well because subsequent calls to Sort takes geometrically decreasing time.

7 An Algorithm with Time Complexity $O(n(\log\log n)^{1.5})$

In Thorup(1998) he builds an exponential search tree and associates buffer $B(v)$ with each node v of the tree. He defines that a buffer $B(v)$ is over-full if $|B(v)| > d(v)$, where $d(v)$ is the number of children of v. Our modification on Thorup's approach is that we define $B(v)$ to be over-full if $|B(v)| > (d(v))^2$. Other aspects of Thorup's algorithm are not modified. Since a buffer is flushed

(see Thorup's definition (Thorup 1998) only when it is over-full, using our modification we can show that the time for flush can be reduced to $|B(v)|\sqrt{\log\log n}$. This will give the $O(n(\log\log n)^{1.5})$ time for sorting by Thorup's analysis (Thorup1 1998).

The flush can be done in theory by sorting the elements in $B(v)$ together with the set $D(v)$ of keys at v's children. In our algorithm this theoretical sorting is done as follows. First, for each integer in $B(v)$, execute $\sqrt{\log\log n}$ steps of the binary search on the dictionary built in Section 3 of (Thorup 1998). After that we have converted the original theoretical sorting problem into the problem of sorting $|B(v)|$ integers (come from $B(v)$ and denoted by $B'(v)$) of $\log m/2^{\sqrt{\log\log n}}$ bits with $d(v)$ integers (coming from $D(v)$ and denoted by $D'(v)$) of $\log m/2^{\sqrt{\log\log n}}$ bits. Note that here a word has $\log m$ bits. Also note that $|B(v)| > (d(v))^2$ and what we needed is to partition $B'(v)$ by the $d(v)$ integers in $D'(v)$ and therefore sorting all integers in $B'(v) \cup D'(v)$ is not necessary. By using the nonconservative version of the algorithm Sort we can then partition integers in $B'(v)$ into sets such that the cardinality of a set is either $< \sqrt{d(v)}$ or all integers in the set are equal. The partition maintains that for any two sets all integers in one set is larger than all integers in anther set. Because we used the nonconservative version of Sort. The time complexity is $|B(v)|\sqrt{\log\log n}$. Then each integer in $D(v)$ can find out which set it falls in. Since each set has cardinality no larger than $\sqrt{d(v)}$ the integers in $D(v)$ can then further partition the sets they fall in and therefore partitioning $B(v)$ in an additional $O(|B(v)|)$ time. Overall the flush thus takes $O(|B(v)|\sqrt{\log\log n})$ time. By the analysis in Thorup(1998) the time complexity for sorting in linear space is thus $O(n(\log\log n)^{1.5})$.

Theorem 7: n integers in $\{0, 1, ..., m-1\}$ can be sorted in $O(n(\log\log n)^{1.5})$ time and linear space.

8 An Algorithm with Time Complexity $O(n \log\log n \log\log\log n)$

The following results are known for nonconservative sorting in linear space.

Lemma 9 (Han and Shen 1999): n integers can be sorted in linear time and space with nonconservative advantage $\log n$.

Theorem 8 (Result in Section 6.): n integers can be sorted into $c\sqrt{n}$ sets $S_1, S_2, ..., S_{c\sqrt{n}}$, where $c \geq 1$ is a constant, such that each set has no more than $\sqrt{n}$ integers and all integers in set S_i are no larger than any integer in set S_j if $i < j$, in time $O(n \log\log n/\log t)$ and linear space with nonconservative advantage $t \log\log n$.

We have $t \log\log n$ nonconservative advantage. The $\log\log n$ nonconservative advantage is used to remove the $\log\log n$ factor from the time complexity of algorithm Sort as we did in Section 5. The other nonconservative advantage t is used to reduce the time complexity of sorting by a factor

of $\log t$ as we did in Section 6.

We use signature sorting (A. Andersson, T. Hagerup, S. Nilsson, R. Raman 1995) to accomplish multi-dividing. We adapt signature sorting to work for us as follows. Suppose we have a set S_1 of t integers already sorted as $a_1, a_2, ..., a_t$ and we wish to use the integers in S_1 to partition a set S_2 of $n > t$ integers $b_1, b_2, ..., b_n$ to $t+1$ sets $S_{20}, S_{21}, ...S_{2t}$ such that all integers in S_{2i} are no larger than any integer in S_{2j} if $i < j$ and for any $c \in S_{2i}$ and $d \in S_{2(i+1)}$ we have $c \leq a_i \leq d$. Suppose h $(2 \log n)$-bit integers can be stored in one word. We first cut the bits in each a_i and each b_i into h 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 hashed values in one word. Let a_i' be the hashed word corresponding to a_i and b_i' be the hashed word corresponding to b_i. We view each hashed word as an integer and sort all these hashed words. As a results a_i''s partition b_i''s into $t+1$ sets $S_0', S_1', ..., S_t'$. They are ordered as $S_0', a_1', S_1', ..., a_t', S_t'$, where all integers in S_i' are no smaller than a_i' and no larger than a_{i+1}'. Let $b' \in S_j'$ then we simply compare b with a_j and a_{j+1} to determine the longest common prefix of bits between b and a_i's. In this way we determine the least significant integer (segment) in a_i's where b "branches out". That is, we cut the number of bits in b_i's to $(1/h)$-th of its original length.

We use a modified version of the exponential search tree as in Andersson (Andersson 1996) and Thorup (Thorup 1998). At the top of the tree is the root and the root has n^c children, where $0 < c \leq 1/2$ is a suitably chosen constant, and each subtree rooted at each child of the root has n^{1-c} nodes and it is recursively defined. Such a tree has $\log \log n / \log(1/(1-c))$ levels. We number the levels top down and therefore the root is at level 0. We will group levels into layers. Layer i contains level $i/\log(1/(1-c))$ through level $(i+1)/\log(1/(1-c)) - 1$. Thus at the top level of layer i there are $n^{(2^i-1)/2^i}$ nodes. As we have mentioned that Raman's method (Raman 1996) to obtain the hash funciton requires $O(n^2 b)$ time, where b is the number of bits of an integer. When $b > n$ we can simply use Andersson's sorting algorithm (Andersson 1996) to sort in $O(n \log \log n)$ time. Note that when $b < n$ in the top layers in the exponential search tree we can still use Raman's hash function, but at layer l where $n^{1/2^{l-1}} > b \geq n^{1/2^l}$ we have to replace each exponential search subtree rooted at the top level of this layer (let this level be level L) by a set. Each such set is sorted by using Andersson's sorting algorithm.

We modify Thorup's definition of over-full for a buffer $B(v)$. We define $B(v)$ to be over-full if $|B(v)| > (d(v))^2$, where $d(v)$ is the number of children of v. The consideration of this modification is that we can flush buffers faster with our definition of over-full. We shall apply Theorem 8 for the purpose of flushing. The dominating time in using the exponential search tree is the flushing and other aspects such as balancing take overall $O(n \log \log n)$ time, as demonstrated by Andersson

(Andersson 1996) and Thorup (Thorup 1998). For each set at level L we also associate a buffer with the set. Such a set has $n^{1/2^l}$ elements. The buffer associated with such a set is over-full if it contains more than $n^{1/2^l}$ elements.

The flushing operation for buffers at level numbered smaller than L can be accomplished with an algorithm A which sorts n integers into $\sqrt{n}$ sets $S_1, S_2, ..., S_{\sqrt{n}}$ such that all integers in S_i is no larger than any integer in S_j if $i < j$. Because we can sort $B(v)$ together with the children of v (call this set C) into $d(v)$ sets with algorithm A. Each integer a in C now falls into one set S_i. Just compare a with all integers in S_i will determine which integers are smaller than a and which integers are larger than a. Do this for all integers in C takes $(d(v))^2 = |B(v)| < T_A$ time, where T_A is the time of running A. This accomplishes the flushing and it takes T_A time.

To accomplish sorting we need to reduce the bits in integers. This is done by perfect hashing in Andersson (Andersson 1996) and Thorup (Thorup 1998). If each integer has b bits we can store, for each vertex v in the tree, all its children's most significant $b/2$ bits in a hash table. This allow us to do binary dividing once on the bits of integers (P. van Emde Boas, R. Kaas, E. Zijlstra 1977). If we store all children's most significant $ib/2^{\sqrt{\log\log n}}$ bits in a hash table T_i, $1 \leq i < 2^{\sqrt{\log\log n}}$, we will be able to do binary dividing on the bits for $\sqrt{\log\log n}$ times. Thereafter we have nonconservative advantage $2^{\sqrt{\log\log n}}$. We now apply Theorem 8 to sort in $O(n\sqrt{\log\log n})$ time. By the reasoning of above paragraph we have accomplished the flushing operation for one layer. Because the exponential search tree has $O(\log\log n)$ layers, we obtain a linear space sorting algorithm with time complexity $O(n(\log\log n)^{3/2})$. This is the algorithm presented in Section 7 and in (Han 2000).

To speed up the above algorithm we note that we can apply multi-dividing to gain nonconservative advantage. Assume that, for sorting n integers, each word has $a\log n(\log\log n)^2$ bits. We cut each word into $\sqrt{a}\log\log n$ integers and hash these $\sqrt{a}\log\log n$ integers in the word to get a total of $2\sqrt{a}\log\log n\log n$ bits hash value. To sort these words of hashed values we have nonconservative advantage of $\sqrt{a}\log\log n/2$ and therefore the sorting can be done in $O(n\log\log n/\log a)$ time by Theorem 8. Thus we have accomplished multi-dividing and have reduced the number of bits in a word to $\log m/(\sqrt{a}\log\log n)$. We now apply Theorem 8 directly to sort the resulting integers (each having $\log m/(\sqrt{a}\log\log n)$ bits stored in the word of $\log m$ bits) in time $O(n\log\log n/\log a)$ because we have nonconservative advantage $\sqrt{a}\log\log n$. This gives an $O(n\log\log n/\log a)$ time algorithm for sorting.

If the word containing an integer has $a\log n\log\log n$ bits where $1 \leq a < \log\log n$, we will cut each word into a integers. After hashing we have $2a\log n$ bits hash value. Sorting these hashed value we have nonconservative advantage $\log\log n$ and therefore the sorting can be done in $O(n\log\log n)$

time. After sorting we reduced each integer to contain $\log n \log\log n$ bits. Therefore these integers can be sorted in $O(n \log\log n)$ time by radix sorting.

If the word containing an integer has $a \log n$ bits, where $1 \leq a < \log\log n$, then we simply use radix sort to sort these integers in $O(n \log\log n)$ time.

We observe that at layer $i > \log\log\log n$ each sorting problem size is $n^{1/2^i}$. Therefore the hashed value has only $2\log n/2^i$ bits. Each word has $\log m$ bits. Thus with regard to hashed values we have nonconservative advantage of $2^{i-1}\log m/\log n \geq 2^{i-1-\log\log\log n}\log n \log\log n/(\log n/\log\log n) \geq 2^{i-\log\log\log n}(\log\log n)^2$ (we assumed that $\log m \geq \log n \log\log n$ for otherwise we simply apply radix sorting for the whole sorting process). Take the value of a in the above paragraph to be $\sqrt{2^{i-1-\log\log\log n}}$ and the time for layer i in the exponential search tree is $O(n\log\log n/(i - \log\log\log n))$. Summing over all layers we obtain time complexity $O(n\log\log n\log\log\log n)$.

For layers $i \leq \log\log\log n$ assuming $\log m \geq \log n \log\log n$ we have $\log\log n \leq 2^{i-1}\log m/\log n \leq \log n\log\log n/(\log n/\log\log n) \leq (\log\log n)^2$. Thus each layer will take $O(n\log\log n)$ time. Summing over $\log\log\log n$ layers the total time is $O(n\log\log n\log\log\log n)$.

If $\log m \leq \log n\log\log n$ we simply sort by using radix sorting.

The flushing at level L is simply done by using Andersson's sorting algorithm.

Thorup (Thorup 1998) also define a node in the exponential search tree *dirty* if the number of integers in the subtree has been doubled. When a node is dirty Thorup (also Andersson 1996) cleans the subtree rooted at the node. This is done by sorting all integers in the subtree. We need modify this operation. When a node is dirty we do not sort sets at level L. A set at level L is sorted only when its buffer is over-full. We do the cleaning by rebuilding the exponential search tree rooted at the dirty node and above level L. In this way we avoid the repeated cleaning of the sets at level L and keep the cleaning operation to within time complexity $O(n\log\log n)$.

Theorem 9: n integers can be sorted in linear space in $O(n\log\log n\log\log\log n)$ time.

Now we consider a special case: sort n integers in range $\{0, 1, ..., m-1\}$ satisfying $\log m \geq (\log n)^{2+\epsilon}$, $0 < \epsilon < 1$ is a constant.

In this case with respect to hashed values the nonconservative advantage is $\log n$ even if $\log^\epsilon n$ hashed values are stored in one word. Using signature sort in linear time we reduce the bits of an integer to $\log m/\log^\epsilon n$. Repeat this process for $1/\epsilon$ times we reduce the bits in an integer to $\log m/\log n$. Thereafter we can sort in linear time by Lemma 9. Thus each layer takes $O(n)$ time. Because there are $\log\log n$ layers the time for the sorting algorithm is $O(n\log\log n)$.

Theorem 10: n integers in $\{0, 1, ..., m-1\}$ can be sorted in $O(n\log\log n)$ time and linear space provided $\log m \geq (\log n)^{2+\epsilon}$.

Previously Andersson (Andersson 1996) showed that when $\log m \geq n^{\epsilon}$, $0 < \epsilon < 1$, sorting can be done in $O(n \log\log n)$ time. Our result substantially enlarged this range in that we require only $\log m \geq (\log n)^{2+\epsilon}$.

9 Conclusions

We have presented improved linear space integer sorting algorithm. We are approaching the time bound in the nonlinear space case of $O(n \log\log n)$ (A. Andersson, T. Hagerup, S. Nilsson, R. Raman 1995, Y. Han, X. Shen 1995). It seems likely that the current time bound of $O(n \log\log n)$ in the nonlinear space case will be eventually achieved in the linear space with a deterministic algorithm.

Acknowledgement

The author wishes to thank the referees for their careful reviewing and constructive comments on this paper.

References

M. Ajtai, J. Komlós, E. Szemerédi (1983), *Sorting in $c \log n$ parallel steps*, Combinatorica, 3, pp. 1-19(1983).

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

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

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

P.C.P. Bhatt, K. Diks, T. Hagerup, V.C. Prasad, T. Radzik, S. Saxena (1991), *Improved deterministic parallel integer sorting*, Information and Computation **94**, 29-47(1991).

P. van Emde Boas, R. Kaas, E. Zijlstra (1977), *Design and implementation of an efficient pri-*

ority queue, Math. Syst. Theory **10** 99-127(1977).

R. Cole (1987/88), *An optimally efficient selection algorithm*, Information Processing Letters, **26**, 295-299(1987/88).

A. Dessmark, A. Lingas (1998), *Improved Bounds for Integer Sorting in the EREW PRAM Model*, J. Parallel and Distributed Computing, **48** 64-70(1998).

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

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

T. Hagerup (1987), *Towards optimal parallel bucket sorting*, Inform. and Comput. **75**, pp. 39-51(1987).

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

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

Y. Han, X. Shen (1995), *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).

Y. Han, X. Shen (1999), *Parallel integer sorting is more efficient than parallel comparison sorting on exclusive write PRAMs.* Proc. 1999 Tenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'99), Baltimore, Maryland, 419-428(January 1999).

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

C. P. Kruskal, L. Rudolph, M. Snir (1990), *A complexity theory of efficient parallel algorithms*,

Theoret. Comput. Sci., **71**, 1, 95-132(1990).

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

P. B. Miltersen (1994), *Lower bounds for union-split-find related problems on random access machines*, Proc. 26th STOC, 625-634(1994).

S. Rajasekaran and J. Reif (1989), *Optimal and sublogarithmic time randomized parallel sorting algorithms*, SIAM J. Comput. **18**, 3, pp. 594-607(1989).

S. Rajasekaran and S. Sen (1992), *On parallel integer sorting*, Acta Informatica 29, 1-15(1992).

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

M. Thorup (1997), *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).

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

R. Vaidyanathan, C.R.P. Hartmann, P.K. Varshney (1993), *Towards optimal parallel radix sorting*, Proc. 7th International Parallel Processing Symposium, pp. 193-197(1993).

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