PARALLEL INTEGER SORTING IS MORE EFFICIENT THAN PARALLEL COMPARISON SORTING ON EXCLUSIVE WRITE PRAMS[*]

YIJIE HAN[†] AND XIAOJUN SHEN[‡]

Abstract. We present a significant improvement for parallel integer sorting. On the EREW PRAM our algorithm sorts n integers in the range $\{0, 1, ..., m-1\}$ in time $O(\log n)$ with $O(n\sqrt{\frac{\log n}{k}})$ operations using word length $k\log(m+n)$, where $1 \le k \le \log n$. In this paper we present the following four variants of our algorithm.
(1). The first variant sorts integers in $\{0, 1, ..., m-1\}$ in time $O(\log n)$ and in linear space with $O(n)$ operations using word length $\log m \log n$.
(2). The second variant sorts integers in $\{0, 1, ..., n-1\}$ in time $O(\log n)$ and in linear space with $O(n\sqrt{\log n})$ operations using word length $\log n$.
(3). The third variant sorts integers in $\{0, 1, ..., m-1\}$ in time $O(\log^{3/2} n)$ and in linear space with $O(n\sqrt{\log n})$ operations using word length $\log(m+n)$.
(4). The fourth variant sorts integers in $\{0, 1, ..., m-1\}$ in time $O(\log n)$ and space $O(nm^{\epsilon})$ with $O(n\sqrt{\log n})$ operations using word length $\log(m+n)$.

Our algorithms can then be generalized to the situation where the word length is $k\log(m+n)$, $1 \le k \le \log n$.

Key words. Algorithms, analysis of algorithms, bucket sorting, conservative algorithms, design of algorithms, integer sorting, parallel algorithms.

AMS subject classifications. 68P10, 68Q22, 68Q25

1. Introduction.

Sorting is a classical problem which has been studied by many researchers. For elements in an ordered set comparison sorting can be used to sort the elements. It is well known that comparison sorting has time complexity $\theta(n\log n)$. In the case where a set contains only integers both comparison sorting and integer sorting can be used to sort the elements. Since elements of a set are usually represented by binary numbers in a digital computer, integer sorting can, in many cases, replace comparison sorting. The only known time lower bound for integer sorting is the trivial linear bound of $\Omega(n)$. Radix sorting does demonstrate $O(n)$ upper bound for sorting n integers in the range $\{0, 1, ..., n^t - 1\}$, where t is a constant. Researchers worked hard trying to show that for integers in any range integer sorting can outperform comparison sorting[4, 13, 19, 21]. Fredman and Willard first showed [13] that n integers in any range can be sorted in $O(n\sqrt{\log n})$ time, thereby demonstrating that in the sequential case integer sorting is more efficient than comparison sorting. However, prior to this paper no deterministic parallel integer sorting algorithm outperformed the lower bound for parallel comparison sorting on any parallel computation models (Detailed explanation is given below). We show, for the first time, that parallel integer sorting is more efficient than parallel comparison sorting on the exclusive write PRAMs.

[*]Preliminary version of this paper is published in Proc. 1999 Tenth Annual ACM-SIAM Symposium on Discrete Algorithms.

[†]School of Interdisciplinary Computing and Engineering, University of Missouri at Kansas City, 5100 Rockhill Road, Kansas City, MO 64110.(`hanyij@umkc.edu`).

[‡]School of Interdisciplinary Computing and Engineering, University of Missouri at Kansas City, 5100 Rockhill Road, Kansas City, MO 64110.(`shenx@umkc.edu`).

The parallel computation model we use is the PRAM model[20] which is used widely by parallel algorithm designers. Usually three PRAM models are used, the EREW (Exclusive Read Exclusive Write) PRAM, the CREW (Concurrent Read Exclusive Write) PRAM and the CRCW (Concurrent Read Concurrent Write) PRAM[20]. In a PRAM model a processor can access any memory cell. On the EREW PRAM simultaneous access to a memory cell by more than one processor is prohibited. On the CREW PRAM processors can read a memory cell simultaneously, but simultaneous write to the same memory cell by several processors is prohibited. On the CRCW PRAM processors can simultaneously read or write to a memory cell. The CREW PRAM is a more powerful model than the EREW PRAM. The CRCW PRAM is the most powerful model among the three variants.

Parallel algorithms can be measured either by their time complexity and processor complexity or by their time complexity and operation complexity which is the time processor product. A parallel algorithm with small time complexity is regarded as fast while a parallel algorithm with small operation complexity is regarded as efficient.

In order to outperform parallel comparison sorting on the exclusive write PRAM models (i.e. CREW PRAM and EREW PRAM) one has to exhibit a parallel algorithm which matches the time lower bound for parallel comparison sorting algorithms and outperform the operation lower bound for parallel comparison sorting algorithms. Note that we cannot outperform the time lower bound (only to match it) because on the CREW and EREW PRAMs the time lower bounds for parallel comparison sorting and for parallel integer sorting are the same, namely $\Omega(\log n)$[11]. The operation lower bound for parallel comparison sorting is $\Omega(n \log n)$ due to the time lower bound for sequential comparison sorting. We explain below that known parallel integer sorting algorithms failed to outperform the lower bound for parallel comparison sorting.

1. Parallel algorithms are known [2, 4, 12, 19, 25] to have operation complexity of $o(n \log n)$ when they are running slower than the $\theta(\log n)$ time lower bound for parallel comparison sorting. But they failed to have $o(n \log n)$ operations when acheiving the time lower bound. For example, the CREW algorithm given in [2] (the best prior to this paper) has operation complexity $O(n\sqrt{\log n})$ when running at time $O(\log n \log\log n)$. But the time lower bound for comparison sorting on the CREW PRAM is $\Omega(\log n)$[11]. It is not clear how to make the algorithm in [2] to run in $O(\log n)$ time. Also the CRCW algorithm in [4, 19] has operation complexity $O(n \log\log n)$ when running at time $O(\log n)$. But the time lower bound for comparison sorting on the CRCW PRAM using polynomial number of processors is $\Omega(\log n/\log\log n)$[6].

2. Parallel algorithms are known [2, 9, 29] that have operation complexity $o(n \log n)$ running at time lower bound for parallel comparison sorting when sorting on small integers. They fail to outperform parallel comparison sorting when sorting on large integers. For example, the previous best results in [2, 12] showed that n integers in the range $\{0, 1, ..., 2^{O(\sqrt{\log n})}\}$ can be sorted on the EREW PRAM in $O(\log n)$ time and linear operations. However, no previous deterministic algorithms showed that n integers larger than $2^{O(\sqrt{\log n})}$ can be sorted in $O(\log n)$ time with $o(n \log n)$ operations on exclusive write PRAMs.

3. Parallel algorithms are known [4, 16] to outperform parallel comparison sorting by using a nonstandard word length (word length is the number of bits in each word). But they fail to outperform on a standard PRAM where word length is bounded by $O(\log(m + n))$. For example in [4] it is shown that sorting n integers in the range $\{0, 1, ..., m - 1\}$ can be done in $O(\log n)$ time with $O(n)$ operations on the EREW

PRAM with a word length of $O((\log n)^{2+\epsilon} \log m)$. The use of extra bits in word length in parallel integer sorting is generally regarded as excess. Note that even in this case (use nonstandard word length) our results are better than all previous results.

In this paper we show for the first time that on the exclusive write PRAMs parallel integer sorting is more efficient than parallel comparison sorting. For sorting n integers in the range $\{0, 1, ..., m-1\}$ our algorithm runs in $O(\log n)$ time with operation complexity $O(n\sqrt{\frac{\log n}{k}})$ when using word length $k \log(m+n)$, where $1 \leq k \leq \log n$. When $k=1$ our algorithm uses standard word length $\log(m+n)$ and runs in $O(\log n)$ time (which is also the lower bound for integer sorting on the CREW and EREW PRAM and which matches the time lower bound for parallel comparison sorting on the CREW and EREW PRAM) with $O(n\sqrt{\log n})$ operations (while parallel comparison sorting has a lower bound $\Omega(n \log n)$ for the operation complexity due to the sequential time complexity lower bound). This algorithm outperforms parallel comparison sorting on the CREW and EREW PRAMs.

There are many previous results on parallel integer sorting [2, 4, 9, 12, 15, 16, 19, 22, 24, 25, 26, 28, 29]. We give a brief comparison of our results with the previous results.

An important parameter in integer sorting is the word length w which is the number of bits in a word. Much effort has been spent toward finding good integer sorting algorithms which are conservative in the sense that they do not use extra bits. According to Kirkpatrick and Reisch[21] an integer sorting algorithm sorting n integers in the range $\{0, 1, ..., m-1\}$ is said to be conservative if the word length is bounded by $O(\log(m+n))$. Significant progress has been made recently in this regard. Andersson et al. [4] and Han and Shen[19] showed conservative integer sorting algorithms that sort n integers in the range $\{0, 1, 2, ..., m-1\}$ in $O(\log n)$ time with $O(n \log\log n)$ operations on the CRCW PRAM. This also implies a conservative sequential algorithm with $O(n \log\log n)$ time. Although much progress has been made on parallel integer sorting on the CRCW PRAM[4, 9, 15, 19] which allows simultaneous read and write to shared memory cells, significant difficulties exist when parallel integer sorting algorithms are to be designed on PRAMs which do not allow simultaneous write.

Consider the problem of sorting n integers in the range $\{0, 1, ..., n-1\}$ which is the most important and standard case. Previous best conservative parallel algorithms running in $O(\log n)$ time on CREW and EREW PRAMs use $O(n \log n)$ operations. Rajasekaran and Sen[25], Albers and Hagerup[2] and Dessmark and Lingas[12] were able to reduce the number of operations to $o(n \log n)$ on the CREW PRAM and EREW PRAM but the running time must be over $O(\log n)$. Currently the best result due to Albers and Hagerup[2] sorts in $O(\log n \log\log n)$ time with $O(n\sqrt{\log n})$ operations on the CREW PRAM. On the EREW PRAM the algorithms in [2, 25] have $O(\log n \log\log n)$ time complexity with $O(n \log n / \log\log n)$ operations. Very recently Dessmark and Lingas presented an improved EREW algorithm[12] which needs $O(\log^{3/2} n)$ time with $O(n\sqrt{\log n})$ operations. Thus in regard to the best previous results one cannot sort better than the comparison sorting algorithm[1, 10] (which uses $O(n \log n)$ operations) if he is to sort as fast as the comparison sorting algorithm (using $O(\log n)$ time) on the CREW and EREW PRAMs.

In this paper we significantly improve on this situation (i.e. sorting n integers in the range $\{0, 1, ..., n-1\}$). Our EREW PRAM algorithm sorts in $O(\log n)$ time with $O(n\sqrt{\log n})$ operations. Thus our algorithm uses the same number of operations ($O(n\sqrt{\log n})$) as the algorithm by Albers and Hagerup[2] and by Dessmark and

Lingas[12] while our algorithm runs faster (in $O(\log n)$ time) than their algorithm (in $O(\log n \log\log n)$ time on the CREW PRAM and in $O(\log^{3/2} n)$ time on the EREW PRAM). Thus our EREW algorithm is faster by a factor of $\log\log n$ than the previous best CREW algorithm and is faster by a factor of $\log^{1/2} n$ than the previous best EREW algorithm.

Now consider the problem of sorting n intergers in the range $\{0, 1, 2, ..., m-1\}$. All previous EREW and CREW conservative algorithms[2, 12, 22, 25, 29] require $O(n \log n)$ operations when m is large, even when the time complexity is allowed to polylogarithmic of n. Actually the number of operations of best previous results is larger than $O(n \log n)$, however, we could assume that these algorithms switch to comparison sorting when m is at certain threshold value. Our result is the first which sorts arbitrarily large integers with $o(n \log n)$ operations. Our EREW integer sorting algorithm sorts in $O(\log n)$ time with $O(n\sqrt{\log n})$ operations. This is for arbitrarily large values of m.

We also present an algorithm (Theorem 4.1) which sorts integers in $\{0, 1, ..., m-1\}$ with $O(\log^{3/2} n)$ time and $O(n\sqrt{\log n})$ operations and it runs in linear space. Previously, Dessmark and Lingas[12] achieved this performance only for sorting integers in the range $\{0, 1, ..., n^k\}$ for a constant k.

We now turn to nonconservative integer sorting. Consider the problem of sorting n integers in the range $\{0, 1, 2, ..., m-1\}$ on a computer with word length w. Hagerup and Shen[16] showed that if $w = \Omega(n \log n \log m)$ the sorting can be done in linear space and in $O(n)$ sequential time or in $O(\log n)$ time on a EREW PRAM with $O(n)$ operations. Later Albers and Hagerup[2] and Andersson et al. [4] improved on the word length. Albers and Hagerup[2] showed that with $w = O(\log n \log\log n \log m)$ the sorting can be done in linear space and in $O(\log^2 n)$ time with $O(n)$ operations on the EREW PRAM. The result of Andersson et al.[4] show that the sorting can be done in linear space and in $O(\log n)$ time with $O(n)$ operations on the EREW PRAM with a word length of $O((\log n)^{2+\epsilon} \log m)$. Dessmark and Lingas[12] showed that the sorting can be done on the EREW PRAM in linear space and in $O(\log n \log\log n)$ time and $O(n)$ operations with a word length of $O(\log m \log n)$. In this paper we improve on all these previous results. We show that the sorting can be done in $O(nm^{\epsilon})$ space and in $O(\log n)$ time with $O(n\sqrt{\frac{\log n}{k}})$ operations on the EREW PRAM with a word length of $O(k \log m)$, where k is a parameter satisfying $1 \leq k \leq \log n$. When $k = \log n$ our algorithm shows that the sorting can be done in linear space and in $O(\log n)$ time with $O(n)$ operations. It is this version of the algorithm that outperforms all previous algorithms. We note that the main focus of this paper is to present conservative EREW algorithms for integer sorting. The nonconservative algorithm we designed is to be used as a subroutine in our conservative algorithms, although our nonconservative algorithm improves on best previous results.

Algorithms presented in this paper are deterministic algorithms. These algorithms are stable in the sense that input integers of the same value retain their original relative order in the output.

2. Nonconservative Sorting. We present an EREW algorithm using word length $O(\log n \log m)$ to sort n integers in the range $\{0, 1, ..., m-1\}$ in $O(\log n)$ time with $O(n)$ operations. The input numbers are assumed in an array. This EREW algorithm is based on the AKS sorting network[1], Leighton's column sort[23], Albers and Hagerup's test bit technique[2] and the Benes permutation network[7, 8].

AKS sorting network [1] has $O(\log n)$ stages. Each stage has no more than $n/2$

comparators. Each comparator has two inputs and two outputs. Each comparator can compare two input numbers and route the smaller number to the left output and the larger number to the right output. When the n input numbers are input to the AKS sorting network and pass through $O(\log n)$ stages of the sorting network they are sorted at the output of the AKS sorting network.

We will use the principle of Leighton's column sort [23]. This principle states that if we put a set S of n numbers into $\log n$ columns with each column containing $n/\log n$ numbers, then the numbers in S will be sorted by a constant number of the following passes: Sort every column and then perform a fixed permutation among the numbers in all columns. Therefore the principle of Leighton's column sort converts the sorting on n numbers to the sorting on $n/\log n$ numbers (there are $\log n$ columns of them and each of them has to be sorted) and fixed permutations among the n numbers. A fixed permutation is a permutation known before program execution. It does not depend on the value of the input numbers. In Leighton's column sort these permutations are shuffle, unshuffle and shift. Note also that if the principle of column sort is recursively applied we can enlarge the number of columns to n^{ϵ}, where $0 < \epsilon < 1$ is a constant, and the number of passes of sorting on columns and permutation is still a constant.

Albers and Hagerup's test bit technique [2] can be used for packed numbers. When there are k numbers packed into a word, we can use this technique to do pairwise comparison of the numbers in two words and extract the larger numbers into one word and smaller numbers into another word. An example will make this clear. A test bit of 0 or 1 is added between the numbers packed into a word. Suppose we pack three numbers a_1, a_2, a_3 (each containing t bits) into a word as $w_1 = 1a_11a_21a_3$ and pack another three numbers b_1, b_2, b_3 (each containing t bits) into a word as $w_2 = 0b_10b_20b_3$. That is we add a test bit of 0 or 1 between the numbers. By doing $w_3 = w_1 - w_2$ and then applying a mask we can extract out the test bits of w_3. These test bits tells us which number is larger. By using these test bits we can subsequently extract out the larger numbers from w_1 and w_2 and put them into one word. Similarly we can also extract out the smaller numbers from w_1 and w_2 and put them into one word. Note that this operation takes constant time no matter how many numbers are packed into one word.

The Benes permutation network [7, 8] is a network with $O(\log n)$ stages. Each stage has $n/2$ switches. Each switch has two inputs and two outputs. When the switch is unset the left input goes to the left output and right input goes to the right output. When the switch is set the left input goes to the right output and the right input goes to the left output. By setting up the switches in the Benes permutation network any permutation of the input can be realized. We will use butterfly networks (explained below) which also has $O(\log n)$ stages and each stage has $n/2$ switches. Butterfly network can emulate the Benes network and realize any permutation.

LEMMA 2.1. *If the word length is $O(\log n \log m)$ we can pack n integers in $\{0, 1, ..., m-1\}$ into $n/\log n$ words in $O(\log\log n)$ time and $O(n)$ operations on the EREW PRAM.*

Proof. First pack two integers into one word and we have $n/2$ words left. Then pack the integers in every two words into one word. Repeat this for $\log\log n$ times we have $\log n$ integers packed into one word. The time is $O(\log\log n)$. The operation is $n + n/2 + n/4 + n/8 + ... = O(n)$. □

With $\log n$ integers packed into one word we can view the n integers in $n/\log n$ words as forming $\log n$ columns. The i-th column contains the i-th integers in all $n/\log n$ words.

Lemma 2.2. *The $n/\log n$ integers in each of the $\log n$ columns can be sorted in $O(\log n)$ time and $O(n)$ operations on the EREW PRAM.*

Proof. We build an AKS sorting network on these $n/\log n$ words. On the AKS sorting network we compare two words at each internal node of the network using the test bit technique. Thus each node of the AKS sorting network can be used to compare the $\log n$ integers in the word in parallel. At the output of the AKS sorting network we have sorted $\log n$ columns with the i-th column containing i-th integers in all $n/\log n$ words. The time is $O(\log n)$ since there are $O(\log n)$ stages in the AKS sorting network. The operation is $(n/\log n) \cdot O(\log n) = O(n)$ because we have only $n/\log n$ words and each of them goes through $O(\log n)$ stages. $\square$

Lemma 2.3. *A fixed permutation among n integers packed into $n/\log n$ words can be done in $O(\log\log n)$ time and $O(n)$ operations on the EREW PRAM.*

Proof. Simply disassemble the integers in the words so that each word contains one integer. Then apply the permutation. Then reassemble the integers into words. $\square$

"$\log n$" in the paper stands actually for the smallest power of 2 no less than $\log n$. This is achieved without increasing the space to superlinear as follows. Take k to be the smallest integer which is a power of 2 and which is no less than $\log n$. Pack k integers into one word. We obtain $l = \lceil n/k \rceil$ words. We take a number l' which is a power of 2 and which is the smallest number no less than l. We assume that we have l' words. Note that here we do not require that $k = \log(l'k)$. The total number of integers is $l'k \leq 3n$.

For our purpose (because we need to sort integers in a linked list) we also need the following scheme to accomplish the permutation mentioned above. The permutation can also be done by routing the integers through a network N which is the butterfly network in conjunction with a reverse butterfly network(see Fig. 2.1.). For permutations N can be used to emulate the Benes permutation network[7][8]. Each stage of the butterfly network emulates the processor connection along a dimension on the hypercube (i.e. at dimension j numbers at position a and $a\#j$ are input into one switch and output to positions a and $a\#j$ where $a\#j$ is obtained by complementing j-th bit of a). When a and $a\#j$ are in different words then we switch integers between the words (in this case every pair goes into a switch is coming from different word). When a and $a\#j$ are in the same word (because we pack integers into word) we switch integers within words (in this case every pair goes into a switch is coming from the same word) (here we need k to be a power of 2). Therefore each stage of the butterfly network can be done in constant time. Because butterfly network has $O(\log n)$ stages, the permutation can be done in $O(\log n)$ time. Note that since the permutations we performed here are fixed permutations the setting of the switches in the butterfly network can be precomputed.

Theorem 2.4. *n integers in the range $\{0, 1, ..., m-1\}$ can be sorted on the EREW PRAM with word length $O(\log n \log m)$ in $O(\log n)$ time using $O(n)$ operations and $O(n)$ space.*

Proof. By Leighton's column sort we need just apply Lemmas 2.1, 2.2 and 2.3 a constant number of times. $\square$

The principle of Theorem 2.4 can be applied to the case where we can pack more than $\log n$ integers into one word. However, in order to apply a recursive version of Leighton's column sort[23] the number of columns cannot be greater than n^ϵ for a constant $0 \leq \epsilon < 1$. Therefore we cannot pack more than n^ϵ integers into one word and then apply the principle of Theorem 2.4. Note also that we may use more

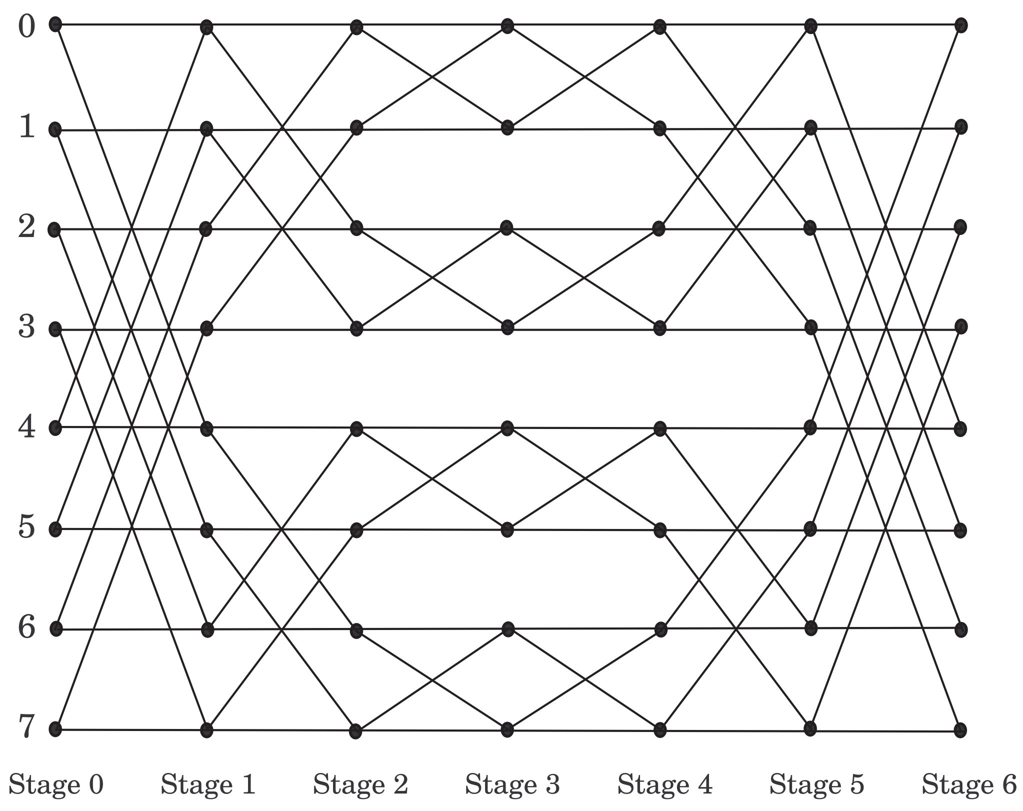

FIG. 2.1. *A permutation network.*

columns than the number of integers packed in one word. For example we may use $\log^2 n$ columns in the column sort even when the number of integers packed in a word is only $\log n$. We will use this fact in the appendix.

The following corollary can now be easily shown.

COROLLARY 2.5. *n integers in the range $\{0, 1, ..., m-1\}$ can be sorted on the EREW PRAM with word length $O(k \log m)$, $1 \leq k \leq \log n$, in $O(\log n)$ time using $O(\frac{n \log n}{k})$ operations and $O(n)$ space.*

This corollary is easily obtained by packing k integers into one word and then by applying Lemmas 2.1, 2.2 and 2.3 and Theorem 2.4.

Note also that the sorting can be made stable by appending address bits to each integer. If $m \leq n$ we use an observation which says that n input integers $a_0, a_1, a_2, ..., a_{n-1}$ can be divided into n/m sets with i-th set containing $a_{im}, a_{im+1}, ..., a_{(i+1)m-1}$ and we need only to sort each set. The results of the sorting on each set can be combined to yield the sorted sequence of input integers. And this combining takes $O(\log n)$ time and linear operations. This observation is made by Rajasekaran and Sen[25]. Algorithms presented in [2] also make use of this observation. Thus the address bits used is always being $\min\{\log n, \log m\}$. To add the address bits we can sort in two passes with each pass sorting on $\log m/2$ bits. Therefore the address bits can be added because the number of address bits now is $\min\{\log n, \log m/2\}$.

3. Sorting Integers in $\{0, 1, ..., n-1\}$. We consider the problem of sorting n integers in the range $\{0, 1, ..., n-1\}$ on the EREW PRAM with word length $O(\log n)$. For our purpose we assume that $\sqrt{\log n}$ is a power of 2. The way this is done is to take the smallest integer which is power of 2 and which is no less than $\sqrt{\log n}$. There is no danger of using superlinear space because this quantity does not determine the space usage.

If input integers with the same value are linked in a linked list in the order they appear in the input, then an additional $O(\log n)$ time and $O(n)$ operations suffice for the sorting. This is because we can use linked list contraction[5] to group integers of the same value together. Because we are sorting integers from $\{0, 1, 2, ..., n-1\}$ we can use bucket sorting. The first integer in each linked list drops itself into bucket. Because there are only n buckets integers dropped into the buckets can be collected in $O(\log n)$ time and $O(n)$ operations. Here the computation is as follows:
1. The first integer (representative integer) in each linked list drops itself into bucket. This is done for all representative integers in parallel in one step. Because different representative integers have different values the dropping operation has no conflicts among integers.
2. Do a parallel prefix computation to pack integers in the buckets into consecutive locations. This will have all integers dropped into buckets sorted.

Our goal, therefore, is to link integers of the same value into a linked list. Initially we put all input integers into one linked list. As the computation proceeds, each linked list is split into several linked lists. When the computation ends, all integers with the same value will be linked into a linked list and integers with different values are in different linked lists.

The basic idea of the sorting algorithm is linked list splitting. Let $a_0, a_1, ..., a_{n-1}$ be the input integers. The algorithm has $\sqrt{\log n}$ stages. In each stage we examine $\sqrt{\log n}$ bits (we say that we reveal $\sqrt{\log n}$ bits). Initially no bits are revealed. In the first stage we reveal the most significant $\sqrt{\log n}$ bits. In the second stage we reveal the next $\sqrt{\log n}$ bits, and so on. We maintain the property that all integers are linked in a linked list if their revealed bits are the same(of the same value). If the revealed bits for two integers are different then the two integers are in different linked lists. Initially all integers are linked into one linked list with a_{i+1} following a_i in the linked list. After the first stage, the input linked list is split into at most $2^{\sqrt{\log n}}$ linked lists because $\sqrt{\log n}$ bits are revealed. After the second stage each linked list further splits itself into at most $2^{\sqrt{\log n}}$ linked lists. And so on.

Now we discuss how each linked list is split in each stage. A linked list is short if it contains less than $2^{4\sqrt{\log n}}$ elements, is long if it contains at least $2^{4\sqrt{\log n}}$ elements. We first group every consecutive S elements(integers) in the linked list into one group. For a short linked list S is the number of total elements in the linked list. For a long linked list S varies from group to group but is at least $2^{4\sqrt{\log n}}$ and no more than $2^{5\sqrt{\log n}}$. We can consider grouping as contracting the S elements into one node and/or as ranking the S elements along the linked list within the group. This grouping can be done by linked list contraction algorithms [5, 17, 18]. We then sort integers in each group in parallel. Because revealed bits for the previous stages for integers in the linked list are identical and because we reveal additional $\sqrt{\log n}$ bits in this stage, we are in fact sorting no more than $2^{5\sqrt{\log n}}$ $\sqrt{\log n}$-bit integers in each group. By our nonconservative sorting algorithm presented in the previous section, the sorting can be done in $O(\sqrt{\log n})$ time and $O(S)$ operations for the group (or $O(n)$ operations for all linked lists). Note that if a short linked list contains too few integers (say t integers) we cannot pack $\sqrt{\log n}$ integers into one word and then apply column sort (see the paragraph immediately below Theorem 2.4). In this situation we pack a smaller number of integers ($\log t$ integers) into a word to facilitate column sorting. For example if a short linked list contains only $\sqrt{\log n}$ integers we then pack $\log(\sqrt{\log n})$ integers into a word and then apply column sort.

If the linked list is short there is only one group in the linked list. The sorting will then enable us to split the linked list into $t \leq 2^{\sqrt{\log n}}$ linked lists such that each linked list split contains all integers whose revealed bits are the same, where t is the number of bit patterns for the revealed bits. Here we note that for short linked list t could be less than $2^{\sqrt{\log n}}$ (for example if the revealed bits for all integers are the same t will be equal to 1).

If the linked list is long we will always split the linked list into exactly $2^{\sqrt{\log n}}$ linked lists no matter how many different bit patterns are revealed by the revealed bits. After sorting in each group, integers in each group are split into $2^{\sqrt{\log n}}$ linked lists. If a bit pattern among the $2^{\sqrt{\log n}}$ bit patterns does not exist in the revealed bits we create a linked list containing only one dummy element representing this pattern. Note that no more than $2^{\sqrt{\log n}}$ dummy elements will be created for each group. For consecutive (neighboring) groups on a long linked list we then join the split linked lists in the groups such that linked lists with the same revealed bits are joined together. With the help of those dummy elements we now have split a long linked list into exactly $2^{\sqrt{\log n}}$ linked lists.

With the existence of dummy elements in the linked list, the splitting process should be modified a little bit. For a short linked list, after the grouping all dummy elements will be eliminated. For a long linked list, the dummy elements will also be eliminated after grouping, but new dummy elements could be created because we need to split each long linked list to $2^{\sqrt{\log n}}$ linked lists.

Since each group on a long linked list has at least $2^{4\sqrt{\log n}}$ elements and since each such a group creates at most $2^{\sqrt{\log n}}$ dummy elements, the total number of dummy elements created in a stage is at most $n/2^{3\sqrt{\log n}}$. Dummy elements generated in a stage are eliminated in the next stage and new dummy elements are generated for the next stage. For a total of $\sqrt{\log n}$ stages the total number of dummy elements generated is no more than $cn\sqrt{\log n}/2^{3\sqrt{\log n}}$ for a constant c. Here constant c may be greater than 1 because as dummy elements are generated we have $n' > n$ elements now. In this situation the number of dummy elements generated in the next stage will be $n'/2^{3\sqrt{\log n'}}$.

Let us estimate the complexity. Since each stage takes $O(\sqrt{\log n})$ time and $O(n)$ operations, for a total of $\sqrt{\log n}$ stages the time complexity of the algorithm is $O(\log n)$ with $O(n\sqrt{\log n})$ operations.

Although we have left several implementation details we hope our presentation in this section can convince most readers that our algorithm works. We therefore give the following theorem. The implementation details are very much ad hoc and several known techniques are adapted to make our implementation fit. The implementation details are described in Appendix A.

THEOREM 3.1. *n integers in the range $\{0, 1, 2, ..., n-1\}$ can be sorted in $O(\log n)$*

time and $O(n)$ space with $O(n\sqrt{\log n})$ operations on the EREW PRAM with word length $O(\log n)$.

4. Sorting Integers in $\{0, 1, ..., m-1\}$. Consider the problem of sorting integers in the range $\{0, 1, ..., m-1\}$. All known conservative CREW and EREW parallel algorithms[2][12][25], even allowing polylogarithmic running time, will eventually use $O(n \log n)$ operations when m is sufficiently large. In this section we present two conservative EREW algorithms which use only $O(n\sqrt{\log n})$ operations no matter how large m is. Our first algorithm runs in $O(\log^{3/2} n)$ time and $O(n)$ space with $O(n\sqrt{\log n})$ operations. This algorithm also serves the purpose of introducing ideas to be used in our second algorithm. The basic ideas used in both algorithms are the same. However, the second algorithm is more complicated than the first algorithm. Our second algorihthm is the only algorithm in this paper which uses more than linear space. This algorithm sorts in $O(\log n)$ time and $O(nm^\epsilon)$, $0 < \epsilon < 1$, space with $O(n\sqrt{\log n})$ operations. We note that the linked list splitting idea presented in the previous section does not apply here and therefore new ideas are needed.

First let us outline our approach. We use *bit* $[i]$ to denote bits $i \log m/\sqrt{\log n}$ through $(i+1)\log m/\sqrt{\log n} - 1$ (bits are counted from the least significant bit starting at 0). $[i : j]$ is used to denote bits $[i], [i+1], ..., [j]$ (or empty if $j < i$). We use $a^{[i]}$ to denote bits $i \log m/\sqrt{\log n}$ through $(i+1)\log m/\sqrt{\log n} - 1$ of a. $a^{[i:j]}$ is used to denote bits $a^{[i]}, a^{[i+1]}, ..., a^{[j]}$ (or empty if $j < i$). To sort n integers with each integer containing $\log m$ bits we could use $\sqrt{\log n}$ passes. The i-th pass, $0 \le i < \sqrt{\log n}$, sorts bit $[\sqrt{\log n} - i - 1]$. Note that we are sorting from high order bits to low order bits. At the beginning of i-th pass the input integers are divided into a collection C of sets such that integers in one set have the same value in bits $[\sqrt{\log n} - i : \sqrt{\log n} - 1]$. In the i-th pass we can sort integers in each $s \in C$ independently and in parallel. We call the sorting problem formed by integers in an $s \in C$ an independent (sorting) problem (or an I-problem for short). The sorting in the i-th pass further subdivides each $s \in C$ into several sets with each set forms an I-problem for the next pass. Note that if a set s_1 resulted from the subdivision (sorting in the i-th pass) of $s \in C$ is a singleton, then the integer $a \in s_1$ needs not to be passed to the next pass because a has been distinguished from other integers and the final rank of a can be determined. When we say an I-problem p we refer to the integers passed from the previous pass to the current pass which form p. When integers in p are sorted in the current pass some of them will be passed to the next pass and these integers are no longer in p. After current pass finishes, p refers to those integers in the singletons which remained and not passed to the next pass. Because in a pass we sort $\log m/\sqrt{\log n}$ bits only while each word has $\log m$ bits, each pass can be computed with $O(n\sqrt{\log n})$ operations (by Corollary to Theorem 2.4). This will give us a total of $O(n \log n)$ operations for the algorithm. To reduce the number of operations, we pipeline all passes. Integers will be passed from the i-th pass to the $(i+1)$-th pass as soon as enough number of integers with the same bits $[\sqrt{\log n} - i - 1 : \sqrt{\log n} - 1]$ are accumulated instead of at the end of the i-th pass. The details will be explained in the following subsection.

4.1. Sort in $O(\log^{3/2} n)$ Time and $O(n\sqrt{\log n})$ Operations. We give an outline which explains the essence of our ideas. Suppose we are sorting n integers. If we pass these n integers through the AKS sorting network [1] we will get $O(\log n)$ time and $O(n \log n)$ operations. This is $O(\log n)$ operations per integer. The reason each integer incurs $O(\log n)$ operations is that it must compare across n integers. If we sort n^t, $0 < t < 1$, integers, then each integer compares across n^t integers, and therefore each integer incurs only $O(t\log n)$ operations. If we take the most significant $\log m/\sqrt{\log n}$ bits out of $\log m$ bits from each integer to form a small integer, we can pack $\sqrt{\log n}$ small integers into one word and therefore we have only $n/\sqrt{\log n}$ words to handle. Sorting these words through the AKS sorting network takes only $O(n\sqrt{\log n})$ operations. However, this sorting may not accomplish the sorting for the input integers. The problems occurs when two integers have the same $[\sqrt{\log n}-1]$ bits, i.e. small integers obtained from them are equal. We treat this problem in this way. We use several levels for sorting. We do not sort all small integers at once. At the level 0 we divide them into groups with each group containing $2^{\sqrt{\log n}}$ integers. In the first stage we sort integers in each group. This entails constant operations for each small integer because each small integer has only $\sqrt{\log n}$ bits. In each next stage we merge $2^{\sqrt{\log n}}$ groups into one group. If the sorting/merging reveals that several small integers are equal we then remove all these small integers and place a dummy to replace them. This dummy has the same value as the small integers removed. Removed integers will participate in sorting at the level 1 where their $[\sqrt{\log n}-2]$ bits are sorted. Since removed small integers no longer participate in sorting in level 0 each of them incurred a few operations. The operations incurred by each integer at levels greater than 0 depends on when it is passed from level 0. If it is passed down at early stages it incurs few operations at level 0 but it will incur more operations at higher numbered levels. If it is passed down at later stages it incurs more operations at level 0 but it will incur less operations at higher numbered levels. This is because if the integer is passed down at later stages there are only a few integers with the same bit value at bits $[\sqrt{\log n}-1]$ as itself, and therefore the I-problem formed at higher numbered levels is of smaller size, and therefore it incurs less operations at higher numbered levels.

In our algorithm the computation is organized into $\sqrt{\log n}$ levels. Each level represents a pass explained in the paragraph before this subsection. There are $\sqrt{\log n}$ stages in each level and stage i_1 at level l_1 is executed concurrently with stage i_2 at level $l_2 > l_1$, where $i_1 - i_2 = l_2 - l_1$. Each stage takes $O(\log n)$ time and $O(n)$ operations. Because there are a total of $2\sqrt{\log n}-1$ stages our algorithm takes $O(\log^{3/2} n)$ time and $O(n\sqrt{\log n})$ operations. The computation at level i, $0 \le i < \sqrt{\log n}$, is to work on bits $[\sqrt{\log n}-i-1]$. We use array $I[0:n-1]$ to represent the n input integer and use $I[i:j]$ to denote $I[i], I[i+1], ..., I[j]$. Although the computation at each level is similar, to describe the computation at an arbitrary level will be too complicated. Instead we first give pseudo codes outlining the overall structure of the algorithm and then describe the computation at levels 0 and 1 and then generalize it to arbitrary levels.

```
$Sort1(A)$
{
a.  for $i = 0$ to $n-1$ do in parallel
b.      { $A[0][i] = A[i]$; /* Put integers at level 0 */

c.  for $k = 0$ to $2\sqrt{\log n}-2$ do /*stage $k$ */
d.  {
e.      if $k < \sqrt{\log n}$ then for $l = 0$ to $k$ do in parallel $B(A,k,l)$; /* level $l$ */
f.      else for $l = k-\sqrt{\log n}+1$ to $\sqrt{\log n}-1$ do in parallel $B(A,k,l)$;
g.  }
}
```

```
$B(A,k,l)$
{
    1.  for $j = 0$ to $n/2^{(k-l+1)\sqrt{\log n}}$ and $i = i(j) = j2^{(k-l+1)\sqrt{\log n}}$ to $(j+1)2^{(k-l+1)\sqrt{\log n}} - 1$ do in parallel
    2.          /* section $j$, here $i(j)$ indicates that this $i$ is coming from section $j$. */
    3.      {
    4.          if $Meet[j][(A[l][i(j)])^{[\sqrt{\log n}-l,\sqrt{\log n}-1]}][(A[l][i(j)])^{[\sqrt{\log n}-l-1]}]$ then
    5.              /* Integers in the same $I$-problem (indicated by $[(A[l][i(j)])^{[\sqrt{\log n}-l,\sqrt{\log n}-1]})$ at level $l$ in section $j$ meet if they have the same $(A[l][i(j)])^{[\sqrt{\log n}-l-1]}$ value. Meet is true if at lease two integers meet.*/
    6.          {
    7.              if $l < \sqrt{\log n} - 1$ then
    8.              {
    9.                  move $A[l][i(j)]$ to $A[l+1][i(j)]$; /* move to next level*/
    10.                 make one integer among the meet become a dummy with value $A[l][i(j)]$, then delete $A[l][i(j)]$;
    11.             }
    12.         }
    13.     }
}
```

The above procedure outlines the overall process of sorting. The Meet operation in procedure B is actually done by sorting and merging. The above procedures give the precise relationship among stage(k), sections(j), levels(l), I-problem(bits $[\sqrt{\log n} - l, \sqrt{\log n} - 1]$), and the value being sorted on(bits $[\sqrt{\log n} - l - 1]$).

The computation at different levels is illustrated in Fig. 4.1 and the Meet operation at different levels is illustrated in Fig. 4.2.

The computation at level 0 is to sort the n input integers by their most significant $\log m/\sqrt{\log n}$ bits. Each stage at level 0 is to merge $2^{\sqrt{\log n}}$ sorted sequences (this is indicated by the Meet operation in line 4 of procedure B). That is, the sorting at level 0 is guided by a complete $2^{\sqrt{\log n}}$-ary tree. Each level of the tree represents the $2^{\sqrt{\log n}}$-way merge in a stage. After stage s and before stage $s+1$ there are $n/2^{(s+1)\sqrt{\log n}}$ sorted sequences (which are the sections in the procedure B). Suppose integer $a^{[\sqrt{\log n}-1]}$ is in the sorted sequence S. a will remain in level 0 of the algorithm as long as there are less than $2^{\sqrt{\log n}}$ integers $b^{[\sqrt{\log n}-1]}$ in S such that $a^{[\sqrt{\log n}-1]} =$

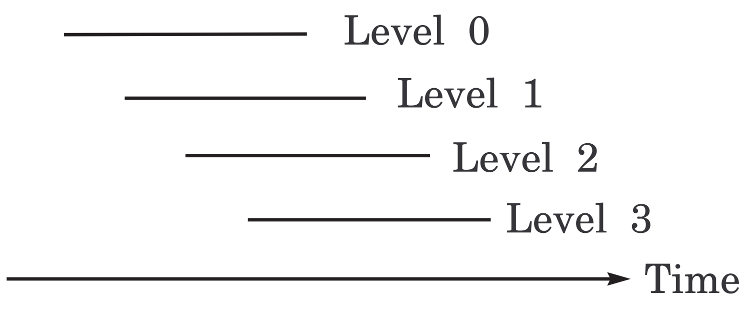

FIG. 4.1. *Computation at different levels is pipelined.*

$b^{[\sqrt{\log n}-1]}$ (which is what we mean by Meet in procedure B) (note that all these integers are now consecutive in memory). Once this condition is not satisfied a will be moved to level 1. Thus one function of level 0 is to group integers $a^{[\sqrt{\log n}-1]}$ and once there are enough integers of the same value grouped together they are sent to level 1. When enough integers of the same value $a^{[\sqrt{\log n}-1]}$ are grouped together in a sorted sequence S and are sent to level 1 we create a dummy with value $a^{[\sqrt{\log n}-1]}$ and place this dummy in S in level 0 to replace the integers sent to level 1. If in a subsequent merge some integers of the same value $a^{[\sqrt{\log n}-1]}$ are grouped together with the dummy (again Meet in procedure B), all these integers (no matter how many) are sent to level 1 and we need only one dummy to represent these integers at level 0. Of course when dummies with same value $a^{[\sqrt{\log n}-1]}$ are grouped together by the merge only one dummy needs to remain while others can be discarded. After the sorting(i.e. all $\sqrt{\log n}$ stages) in level 0 finishes, integers remain in level 0 are those that do not have $2^{\sqrt{\log n}}$ or more input integers with the same $a^{[\sqrt{\log n}-1]}$ value. For integers with the same $a^{[\sqrt{\log n}-1]}$ value in level 0 (there are less than $2^{\sqrt{\log n}}$ of them) we sort them by their whole integer value (not just the most $\log m/\sqrt{\log n}$ bits) by comparison sorting [1][10]. Because each such comparison sorting is on no more than $2^{\sqrt{\log n}}$ elements, the number of operations will be bounded by $O(n\sqrt{\log n})$. After this comparison sorting all integers and dummies at level 0 are sorted. Level 0 has divided integers passed to level 1 into I-problems. Integers a with the same $a^{[\sqrt{\log n}-1]}$ value which are passed to level 1 are in one such I-problem. Now we need to sort integers in each I-problem independently and in parallel.

Now consider the computation at level 1. We consider only one I-problem. The problem is to sort integer a's by $a^{[\sqrt{\log n}-2]}$ value, where the value $a^{[\sqrt{\log n}-1]}$ for all a's are identical. The sorting at level 1 also has $\sqrt{\log n}$ stages and is also guided by a conceptual $2^{\sqrt{\log n}}$-ary tree. However, many of the leaves may be empty because no integer is passed from level 0. Stage s at level 0 and stage $s-1$ at level 1 (and stage $s-i$ at level i) are executed concurrently. Immediately after stage 0 at level 0 all integers a with the same $a^{[\sqrt{\log n}-1]}$ values in $I[i2^{\sqrt{\log n}} : (i+1)2^{\sqrt{\log n}}-1]$ are grouped together, where $0 \le i < n/2^{\sqrt{\log n}}$. If there are integers a in $I[i2^{\sqrt{\log n}}$:

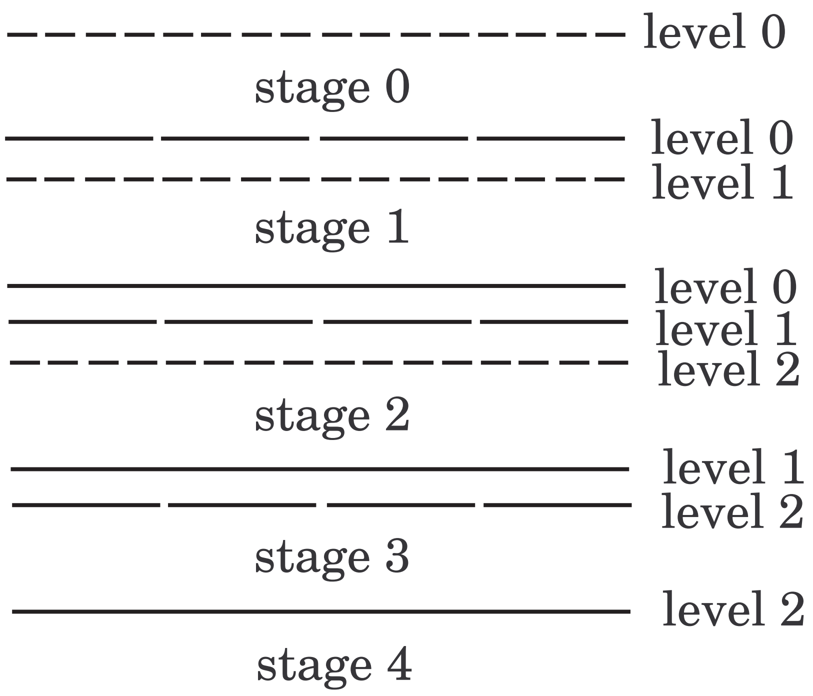

FIG. 4.2. *The merging (Meet) process at different stages.*

$(i+1)2^{\sqrt{\log n}}-1]$ which have the same $a^{[\sqrt{\log n}-1]}$ value and are passed down to level 1 in stage 0 of level 0, these integers are merged (sorted) by the $2^{\sqrt{\log n}}$-way merge on the $a^{[\sqrt{\log n}-2]}$ value in stage 0 at level 1. In stage 1 at level 0 all integers a with the same $a^{[\sqrt{\log n}-1]}$ values in $I[i2^{2\sqrt{\log n}} : (i+1)2^{2\sqrt{\log n}}-1]$ are grouped together, where $0 \leq i < n/2^{2\sqrt{\log n}}$. Consider integers a with the same $a^{[\sqrt{\log n}-1]}$ value in $I[i2^{2\sqrt{\log n}} : (i+1)2^{2\sqrt{\log n}}-1]$. These integers are grouped into a collection C of at most $2^{\sqrt{\log n}}$ groups (which are sections in procedure B) in stage 0 of level 0 (group j coming from $I[j2^{\sqrt{\log n}} : (j+1)2^{\sqrt{\log n}}-1]$, $i2^{\sqrt{\log n}} \leq j \leq (i+1)2^{\sqrt{\log n}}-1$). These $2^{\sqrt{\log n}}$ groups are further grouped into one group G in stage 1 of level 0. If some groups in C are passed down to level 1 at stage 0 of level 0, These passed down groups are sorted in stage 0 at level 1 (which execute in parallel with stage 1 of level 0). Note the relation between sections and levels in procedure B. If there is at least one group passed down to level 1, there will be a dummy at level 0 and therefore all integers in G will be passed down at stage 1 of level 0. By using the dummies at level 0 we will be able to build a linked list to link integers passed down at stage 0 with integers passed down at stage 1. And by executing linked list ranking[5] we can then move all integers in G into consecutive memory locations. Here list ranking takes $O(|G|)$ operations and $O(\log n)$ time. Note that linked list linking and ranking here also maintain the stable property for sorting. Our intention is to do a $2^{\sqrt{\log n}}$-way merge (one way for integers in a group in C) at stage 1 of level 1. However, integers in G which are passed down at stage 1 of level 0 (denote this set by G') are not sorted by $a^{[\sqrt{\log n}-2]}$ and therefore they cannot participate in the $2^{\sqrt{\log n}}$-way merge directly. What we do is to first sort integers in G' by bit $[\sqrt{\log n}-2]$ and then perform the merge. Because G' contains less than $2^{2\sqrt{\log n}}$ integers and because sorting is performed on integers each having $\log m/\sqrt{\log n}$ bits the sorting can be done in $O(\sqrt{\log n})$ time and linear operations by Theorem 2.4.

Thus at level 1 we are forming sorted sequences (sections in procedure B) (sorted by bits $[\sqrt{\log n}-2 : \sqrt{\log n}-1]$) and repeatedly merge the sorted sequences (form larger sections). Suppose integer $a^{[\sqrt{\log n}-2:\sqrt{\log n}-1]}$ is in the sorted sequence S. a will remain in level 1 as long as there are less than $2^{\sqrt{\log n}}$ integers $b^{[\sqrt{\log n}-2:\sqrt{\log n}-1]}$ in S such that $a^{[\sqrt{\log n}-2:\sqrt{\log n}-1]} = b^{[\sqrt{\log n}-2:\sqrt{\log n}-1]}$ (note that all these integers are now consecutive in memory). Once this condition is not satisfied a will be moved to level 2 and we will create a dummy with value $a^{[\sqrt{\log n}-2:\sqrt{\log n}-1]}$ at level 1 to replace the integers moved to level 2. As we did in level 0, for integers a stayed in level 1 and never passed to level 2, there are less than $2^{\sqrt{\log n}}$ integers with the same $a^{[\sqrt{\log n}-2:\sqrt{\log n}-1]}$ values and therefore we can sort them by their whole integer value using parallel comparison sorting after the $(\sqrt{\log n}-1)$-th stage at level 1.

The relation of level 2 to level 1 is the same as that of level 1 to level 0. In general, integers passed to level i are divided to belong to I-problems with each problem containing integer a's with the same $a^{[\sqrt{\log n}-i:\sqrt{\log n}-1]}$ value. In each such problem at level i integers are either sorted at level i (by repeated $2^{\sqrt{\log n}}$-way merge) or passed down to level $i+1$. Integers passed to level $i+1$ are divided at level i into I-problems such that integer a's with the same $a^{[\sqrt{\log n}-i-1:\sqrt{\log n}-1]}$ value are in one I-problem.

There are a total of $2\sqrt{\log n}-1$ stages executed in our algorithm. After these stages and after we use parallel comparison sorting to sort integer a's with the same $a^{[\sqrt{\log n}-i-1:\sqrt{\log n}-1]}$ value at level i, integers at all levels are sorted. We can then build a linked list. For integers and dummies in each I-problem we simply let each element point to the next element. We then "insert" integers sorted in each I-problem p at level i into the position of the corresponding dummy at level $i-1$ by using a pointer from the dummy to point to the first integer in p and using another pointer from the last integer in p to point to the successor of the dummy. We therefore build a linked list for all the integers and these integers are in sorted order in the linked list. After a linked list ranking[5] we have all the integers sorted.

At the end of each stage of our algorithm we use linked list ranking[5] and standard parallel prefix computation[20] to move integers and dummies belonging to each I-problem in consecutive memory locations so that next stage can proceed. For example, integers in an I-problem p at level i need to be packed to consecutive memory locations because some integers in p are passed to level $i+1$. When some integers and dummies in p are grouped into one group by the merging at level i because they have the same $a^{[\sqrt{\log n}-i-1:\sqrt{\log n}-1]}$ value, we build linked list to link the integers at level i with the integers already at level $i+1$(they are represented by the dummies at level i). We use linked list ranking and prefix computation to move these integers in one group into consecutive memory locations. Because linked list ranking and prefix computation can be done in $O(\log n)$ time and $O(n)$ operations they are within the time and the number of operations allocated to each stage. Note here that we generate one dummy for at least $2^{\sqrt{\log n}}$ integers in each stage. Thus for all stages the total number of dummies generated is bounded by $2n\sqrt{\log n}/2^{\sqrt{\log n}}$.

Now we discuss the $x \le 2^{\sqrt{\log n}}$-way merge performed on a collection C of x sorted sequences in each stage at each level. The integers to be merged are in consecutive memory locations and processors can be easily allocated to them. The integers we are considering here have only $\log m/\sqrt{\log n}$ bits while each word has $\log m$ bits. We have to accomplish the merge in $O(\log n)$ time and linear number of operations. If the total number of integers to be merged together is $N < 2^{2\sqrt{\log n}}$ we simply sort them by using Theorem 2.4. Otherwise $N \ge 2^{2\sqrt{\log n}}$ and we sample every $2^{\sqrt{\log n}}$-th integer from each of the $x \le 2^{\sqrt{\log n}}$ sorted sequence (to be merged). If a sequence has no more than $2^{\sqrt{\log n}}$ integers we sample its first and last integers. The total number of sampled integers is no more than $2N/2^{\sqrt{\log n}}$. We sort all sampled integers into one sequence S using parallel comparison sorting[1][10]. We make x copies of S. We then merge one copy of S with one sequence in C. Suppose s_1, s_2, $s_1 \le s_2$, are two consecutive integers in S. Then there are no more than $2^{\sqrt{\log n}}$ integers in each sequence in C which are $\le s_2$ and $\ge s_1$ (for equal integers their order is determined first by the sequence they are in and then by the position they are in the sequence). These integers form a merging subproblem. Because S is merged with each sequence in C, the original merging problem is now transformed into $|S|+1$ ($|S|$ is the number of integers in S) merging subproblems each of them is to merge x subsequences with each subsequence containing at most $2^{\sqrt{\log n}}$ integers (they come from a sequence in C). For each merging subproblem we use Theorem 2.4 to sort all integers in the subproblem.

It can now be checked that the $2^{\sqrt{\log n}}$-way merge in each stage at each level takes $O(\log n)$ time and linear operations. At the end of each stage we use linked list ranking and parallel prefix computation to move integers belonging to each I-problem into consecutive memory locations. These computation takes $O(\log n)$ time and $O(n)$ operations. Therefore each stage takes $O(\log n)$ time and $O(n)$ operations.

THEOREM 4.1. *n integers in the range $\{0, 1, ..., m-1\}$ can be sorted in $O(\log^{3/2} n)$ time and $O(n)$ space with $O(n\sqrt{\log n})$ operations and $O(\log(m+n))$ word length on the EREW PRAM.*

4.2. Sort in $O(\log n)$ Time and $O(n\sqrt{\log n})$ Opertaions. Our second algorithm will run in $O(\log n)$ time with $O(n\sqrt{\log n})$ operations. This algorithm is more complicated. To achieve $O(\log n)$ time we have to allocate only $O(\sqrt{\log n})$ time to each stage. An immediate problem is the following. Consider an I-problem p at level 1. Integers are passed to p at different stages. Suppose several stages have passed and each sorted sequence in p is pretty long. Now in the current stage a set S of integers are passed down from level 0 to p. Although the number of integers in S are few, to merge integers in S with the sorted sequences in p takes long time because a sorted sequence in p contains too many integers. The problem here is that not all integers in p are passed down in one stage, some passed down earlier while others passed down later. If, for example, all integers in p are passed from level 0 at stage 0 (of level 0), then we can merge the integers in p. To avoid the problem that integers are passed down at different stages, we modify our first algorithm (given in Theorem

4.1) as follows.

We append $\log n$ bits to each input integer to indicate the position of each integer in the input. Note that we can assume $\log m > 2\log n$. To put $\log n$ bits into a integer we could sort in two passes with each pass sorting $\log m/2$ bits. Then in each pass we can put $\log n$ bits into each integer. The process of our sorting is not stable. Adding the $\log n$ address bits stablizes the sorting. At each level our algorithm sorts $\log m/\sqrt{\log n}$ bits. In the process of our algorithm execution, an integer a in a sorted sequence at level i will stay in level i as long as the number of integers b in the sorted sequence with $b^{[\sqrt{\log n}-i-1:\sqrt{\log n}-1]} = a^{[\sqrt{\log n}-i-1:\sqrt{\log n}-1]}$ is less than $2^{4\sqrt{\log n}}$. Once this condition is not satisfied, a will be passed to level $i+1$. We keep a dummy at level i to replace the integers passed down to level $i+1$. If there are integers c with $c^{[\sqrt{\log n}-i-1:\sqrt{\log n}-1]} = a^{[\sqrt{\log n}-i-1:\sqrt{\log n}-1]}$ left at level i, they keep grouping as integers at level i are merged. If dummies and integers of the same $[\sqrt{\log n}-i-1:\sqrt{\log n}-1]$ bit values are grouped together we view dummies as smaller than integers and therefore we group dummies with dummies and integers with integers if they have the same value in bits $[\sqrt{\log n}-i-1:\sqrt{\log n}-1]$. Note that here we do not pass integers to level $i+1$ when integers are grouped with dummies. Instead we keep grouping more integers together. Once the new group contains $2^{4\sqrt{\log n}}$ or more integers with the same $c^{[\sqrt{\log n}-i-1:\sqrt{\log n}-1]}$ value all integers in the group will be passed to level $i+1$. Integers remain at a level to the end of the last stage will have less than $2^{4\sqrt{\log n}}$ integers with the same revealed bit values. We can then sort them by their whole integer value by comparison sorting.

We define a grouped subproblem of sorting (G-problem for short). The n input integers are initially in a G-problem p. As computation proceeds, some integers in p will be passed to level 1 and will no longer in p. Some dummies will be created and added to p. *All integers at level $i+1$ which are passed from a G-problem at level i at stage t of level i form a G-problem.*

Consider a G-problem at level i. Integers in different G-problems are sorted independently even if they have the same value in bits $[\sqrt{\log n}-i-1:\sqrt{\log n}-1]$. Integers in a G-problem may have different value in bits $[\sqrt{\log n}-i-1:\sqrt{\log n}-1]$. Thus a G-problem is further divided into I-problems such that each I-problem contains all the integers in the G-problem which have the same bits $[\sqrt{\log n}-i-1:\sqrt{\log n}-1]$. Note that the definition of an I-problem is slightly different than we defined before because now we require integers in an I-problem to be those within the same G-problem. All I-problems in a G-problem can be solved independently.

For the G-problem p at level 0 we execute $(1/2)\log\log n-1$ stages. In the i-th stage we sort every $2^{2^{i+2}\sqrt{\log n}}$ integers by their bit $[\sqrt{\log n}-1]$. That is, in the i-th stage every array $I[j2^{2^{i+2}\sqrt{\log n}},(j+1)2^{2^{i+2}\sqrt{\log n}}-1]$, $0\le j< n/2^{2^{i+2}\sqrt{\log n}}$, is sorted. Integers may be passed to level 1 at different stages and form different G-problems at level 1. Because there are $(1/2)\log\log n-1$ stages at level 0, only $(1/2)\log\log n-1$ G-problems are created at level 1. Consider integers passed to level 1 at the i-th stage which form a G-problem q at level 1. Integers in q with different bit $[\sqrt{\log n}-1]$ are in different I-problems. Consider an I-problem r in q. Because integers are passed at the i-th stage there are at most $S=n/2^{(2^{(i+1)}-4)\sqrt{\log n}}$ integers in r for $i\ge 1$ and there are at most $S=n$ integers in r for $i=0$. We can store the integers in r in an array A of size S as follows. For stage 0 integers in r which are passed from $I[j2^{4\sqrt{\log n}},(j+1)2^{4\sqrt{\log n}}-1]$, $0\le j<n/2^{4\sqrt{\log n}}$, at level 0 are stored

in $A[j2^{4\sqrt{\log n}}, (j+1)2^{4\sqrt{\log n}}-1]$, $0 \le j < n/2^{4\sqrt{\log n}}$. For stage $i > 0$, integers in r which are passed from $I[j2^{2^{i+2}\sqrt{\log n}}, (j+1)2^{2^{i+2}\sqrt{\log n}}-1]$, $0 \le j < n/2^{2^{i+2}\sqrt{\log n}}$, at level 0 are stored in $A[j\frac{2^{2^{i+2}\sqrt{\log n}}}{2^{(2^{(i+1)}-4)\sqrt{\log n}}}, (j+1)\frac{2^{2^{i+2}\sqrt{\log n}}}{2^{(2^{(i+1)}-4)\sqrt{\log n}}}-1]$, $0 \le j < n/2^{2^{i+2}\sqrt{\log n}}$. Note that although all integers in p can be stored in A there may be many blank cells in A with no integers stored in them. Also integers passed from $I[j2^{2^{i+2}\sqrt{\log n}}, (j+1)2^{2^{i+2}\sqrt{\log n}}-1]$ are now in consecutive positions in A and there are at least $2^{4\sqrt{\log n}}$ of them. Integers in each I-problem in q can be stored in an array of size S. We say that the G-problem q has size S.

For each I-problem r in a G-problem of size R we store integers in r in an array A of size R and execute s stages, where s is the minimum integer satisfying $2^{2^{s+1}\sqrt{\log n}} \ge R$. In the i-th stage, $0 \le i < s$, we sort integers in $A[j2^{2^{i+2}\sqrt{\log n}}, (j+1)2^{2^{i+2}\sqrt{\log n}}-1]$, $0 \le j < R/2^{2^{i+2}\sqrt{\log n}}$. As we said before integers are grouped by the sorting and if the number of integers of the same revealed bits is at least $2^{4\sqrt{\log n}}$ they are sent to the next level (they form an I-problem in a new G-problem in the next level). We shall use the algorithm of Theorem 2.4 to do the sorting (detailed to be explained below) and therefore the time expended in the i-th stage is $c2^i\sqrt{\log n}$ for a constant c.

Our sorting algorithm can be summarized in the following coding:

```
$Sort2(A,l,g)$ /*Sort on a set of integers at level $l$ in the $g$-th $G$-problem*/
{
  for ($i=1$; $i \le s$; $i++$) /* $s = \min\{t | 2^{2^{t+2}\sqrt{\log n}} \ge |A|\}$*/
  {
     Divide integers in $A$ to $|A|/2^{2^{i+2}\sqrt{\log n}}$ groups and sort every group in parallel
on $[\sqrt{\log n}-l-1, \sqrt{\log n}-1]$ bits;
     if sets $S$ of integers with the same value on bits $[\sqrt{\log n}-l-1, \sqrt{\log n}-1]$ are
detected do in parallel
       {
         if $|S| \ge 2^{4\sqrt{\log n}}$ { remove $S$ from $A$, replace it with a dummy and call
$Sort2(S, l+1, g \cdot \log\log n + i - 1)$; }
       }
  }
}
```

Note that in $Sort2$ when several sets S's with the same value in bits $[\sqrt{\log n} - l - 1, \sqrt{\log n} - 1]$ are detected in different groups of A and all these sets have size no less than $2^{4\sqrt{\log n}}$, these sets will be moved into one I-problem of a G-problem. The fact that they are moved into the same I-problem is effected by placing them in the same array.

We say that a G-problem (I-problem) p is solved if integers in each I problem at all levels generated from p is sorted. The time complexity for solving a G-problem p can now be expressed as follows. Assume that the size of p is S, p is at level l and the time complexity of solving p is $f(S,l)$ and let $t = \min\{i | 2^{2^{i+2}\sqrt{\log n}} \ge S\}$. We have:

$$f(S,l) = \max\{\max_{i=1}^{t} c2^i\sqrt{\log n} + f(S/2^{(2^{(i+1)}-4)\sqrt{\log n}}, l+1), \\ c\sqrt{\log n} + f(S, l+1)\}$$

$f(S, \sqrt{\log n}) = 0$

Thus the time complexity for solving the G-problem at level 0 is $f(n, 0) = O(\log n)$.

Now let us consider the operation complexity for solving a G-problem. Suppose the integers in an I-problem p is stored in array A. Suppose at the current stage (stage i) p has S integers and dummies (note that many integers may have already passed to the next level at earlier stages). S may be much smaller than the size of A. However, integers are not scattered around in A. Instead they are stored as segments with each segment containing at least $2^{4\sqrt{\log n}}$ integers stored in consecutive memory locations if the segment does not contain a dummy. Thus in the current stage (stage i) we can pack the integers in $A[j2^{2^{i+2}\sqrt{\log n}}, (j+1)2^{2^{i+2}\sqrt{\log n}} - 1]$, $j = 0, 1, 2, ...$, to consecutive memory locations using $O(2^i\sqrt{\log n})$ time and $O(S + d2^i\sqrt{\log n})$ operations, where d is the number of dummies. Since the number of dummies is a fraction of the total number of integers, the total number of operations is linear. After we packed integers we use Corollary to Theorem 12.4 to sort them. The sorting takes $O(2^i\sqrt{\log n})$ time and $O(S2^i)$ operations. This is to say that if an integer a stayed in p until stage i it incurs $O(2^i)$ operations. However, since a is passed to the I-problem q at the next level at stage i, the size of q is the size of p divided by $2^{(2^{(i+1)}-4)\sqrt{\log n}}$ if $i > 0$ and is at most the same as that of p if $i = 0$. Assume that the size of p is S, p is at level l, and the number of operations incurred by an integer a in p is $g(S, l)$. Then:

$$g(S, l) = \max\{\max_{i=1}^{t} c2^i + g(S/2^{(2^{(i+1)}-4)\sqrt{\log n}}, l+1), \\ c + g(S, l+1)\}$$

$g(S, \sqrt{\log n}) = 0$

Thus the number of operations incurred by each integer in the G-problem at level 0 is $g(n, 0) = O(\sqrt{\log n})$. Thus the operation complexity for solving the G-problem at level 0 is $O(n\sqrt{\log n})$.

Below we will discuss how to have all the input integer sorted after we solve the G-problem at level 0.

How many G-problems will be created in the execution of our algorithm? Consider a G-problem p at level l with size S. p executes $\log \frac{\log S}{\sqrt{\log n}} - 1$ stages which generates $\log \frac{\log S}{\sqrt{\log n}} - 1$ G-problems at the next level. The i-th G-problem, $i > 0$, generated has size $S/2^{(2^{i+1}-4)\sqrt{\log n}}$. Let the number of G-problems at all levels which are generated from p be $h(S, l)$. We have:

$$h(S, l) = h(S, l+1) + 1 + \sum_{i=1}^{\log((\log S)/(\sqrt{\log n}))-2} (h(S/2^{(2^{i+1}-4)\sqrt{\log n}}, l+1) + 1),$$

$h(S, \sqrt{\log n} - 1) = 1$

$h(n, 0)$ is the total number of G-problems generated in our algorithm. It is not difficult to see that $h(n, 0) > 2^{\sqrt{\log n}}$. To evaluate the above formula we enlarge it and obtain the following formula:

$$h(S, l) \leq 4 \sum_{i=0}^{(\log S)/(\sqrt{\log n})} h(S/2^{i\sqrt{\log n}}, l+1),$$

$h(S, \sqrt{\log n} - 1) = 1$

which can be rewritten as

$C(j,l) \leq 4\sum_{k=0}^{j} C(k,l+1)$
$C(j,\sqrt{\log n}-1)=1.$

where $C(j,l)=h(2^{j\sqrt{\log n}},l)$.

The total number of groups generated by our algorithm is $h(n,0)$, which is:

$$h(n,0)=C(\sqrt{\log n},0)$$
$$=4\sum_{k_1=0}^{\sqrt{\log n}} C(k_1,1)$$
$$=4^{\sqrt{\log n}}\sum_{k_1=0}^{\sqrt{\log n}}\sum_{k_2=0}^{k_1}\sum_{k_3=0}^{k_2}\cdots\sum_{k_{\sqrt{\log n}}=0}^{k_{\sqrt{\log n}-1}} 1$$
$$\leq 2^{\delta\sqrt{\log n}}, \text{ where } \delta<3.5.$$

Therefore there are no more than $2^{\delta\sqrt{\log n}}$ G-problems generated.

We attach a tag to each integer and dummy to indicate which G-problem it is in. Although the tag can be implemented by an $O(\sqrt{\log n})$-bit integer because there are only $2^{\delta\sqrt{\log n}}$ G-problems, to facilitate the computation of the tag when an integer is passed from a G-problem at level i to another G-problem at level $i+1$ we use an $O(\sqrt{\log n}\log\log\log n)$-bit integer for a tag. If an integer a is in the G-problem p at level i and its tag is t, and it is passed at stage s of level i to another G-problem p_1 at level $i+1$ we form the new tag for a by appending $O(\log\log\log n)$ bits indicating s to t. Dummies created to represent integer passed to the next level inherit the new tag of the integers after they are passed. Thus all integers in a G-problem have the same tag. Dummies in a G-problem may have different tags because they represent integers passed to next level at different stages.

Now consider an I-problem p in a G-problem at level i. As the sorting in p proceeds, integers of same value in bits $[\sqrt{\log n}-i-1:\sqrt{\log n}-1]$ are grouped in each sorted sequence in p. If the number of integers of the same value $a^{[\sqrt{\log n}-i-1:\sqrt{\log n}-1]}$ is no less than $2^{4\sqrt{\log n}}$ these integers will be passed down to a G-problem at the next level. A dummy will be created in p to represent these integers. The dummy has the tag which is the same as the new tag those integers obtained after they passed to the next level. Integers with value $a^{[\sqrt{\log n}-i-1:\sqrt{\log n}-1]}$ left in p will keep grouping as the sorting proceeds. When a new group contains at least $2^{4\sqrt{\log n}}$ integers they will be passed to another G-problem and another dummy will be created for them. All integers in the G-problem which are passed down at a stage form a new G-problem. When integers and dummies of the same value in bits $[\sqrt{\log n}-i-1:\sqrt{\log n}-1]$ are grouped together in p we assume that dummies are smaller than integers and this allows integers to be grouped with integers to form new groups with more than $2^{4\sqrt{\log n}}$ integers. When dummies with the same bit values and same tag are grouped together (they represent integers passed to the same I-problem in a G-problem) all of them but one can be removed. But dummies with different bit values or different tags cannot cancel each other.

Suppose now that integers and dummies in each I-problems are sorted and occupying consecutive memory locations. We need to put all integers in a G-problem together to form a sorted sequence. If we accomplished this we will have only $2^{\delta\sqrt{\log n}}$ sorted sequences left because there are only $2^{\delta\sqrt{\log n}}$ G-problems. We first reduce the number of bits used for a tag to $O(\sqrt{\log n})$ (remember we were using $O(\sqrt{\log n}\log\log\log n)$ bits). This is accomplished easily by sorting all input integers in the input array by their tags. Here we are sorting on $O(\sqrt{\log n}\log\log\log n)$ bits and it can be done by Theorem 3.1. Thereafter we reduce the value for a tag to be within $\{0, 1, ..., 2^{\delta\sqrt{\log n}} - 1\}$ by eliminating unused values. Thus the number of bits used for a tag becomes $\delta\sqrt{\log n}$. Then we sort integer and dummies in an I-problem p at level i by their tag value. Note that integers in p have the same tag value while dummies may have different tags. Note also that integers and dummies in p were sorted by bits $[\sqrt{\log n} - i - 1 : \sqrt{\log n} - 1]$. By sorting on their tags we arrange dummies with the same tag value into consecutive memory location while dummies with the same tag value are still in sorted order by their value of bits $[\sqrt{\log n} - i - 1 : \sqrt{\log n} - 1]$. We then add a pointer for the first and last integers and each dummy in p to point to the dummy at level $i - 1$ which represents the integers in p when they are passed from level $i - 1$ to level i. We now view that all integers and dummies being stored in a big array A (of course many cells A are blanks). We allocate $2^{\delta\sqrt{\log n}}$ arrays $A_0, A_1, ..., A_{2^{\delta\sqrt{\log n}}-1}$ each of them has the same size as A. Array A_i is used to store integers in i-th G-problem (integers tagged with i). For each integer a in A, if a is in $A[k]$ and tagged with i, we now move a to $A_i[k]$ in constant time. If a is the first or last integer in an I-problem (therefore a has a pointer pointing to $A[j]$ which is a dummy at the lower numbered level), a move this pointer to A_i and pointing to $A_i[j]$. For each dummy d in $A[k]$ we make $2^{\delta\sqrt{\log n}}$ copies of it and put one copy in the k-th cell of each A_i, $0 \leq i < 2^{\delta\sqrt{\log n}}$. If the pointer of d (which points to a dummy at the previous level) points to $A[j]$ we copy the pointer to each A_i and make it point to $A_i[j]$, $0 \leq i < 2^{4\sqrt{\log n}}$. Note that because each dummy is created at a level for at least $2^{4\sqrt{\log n}}$ integers, we can allocate $2^{\delta\sqrt{\log n}}$ processors for each dummy. Up to now we have separated integers in different G-problems into different arrays. The integers and dummies in each array A_i together with their pointers form a tree. By using pointer jumping along the Euler tour of the tree[27] we obtain a sorted list of integers in each G-problem. Consider the pointers allocated to each I-problem. If an I-problem does not contain a dummy (i.e. no integers in the I-problem is passed to next level) then the I-problem has at least $2^{4\sqrt{\log n}}$ integers. We need to use two pointers, one at the beginning of the integers and one at the end of the integers for linking this I-problem with other I-problems in the same G-problem. If the I-problem has a dummy we can always allocate three pointers per dummy. Then one pointer is used for the dummy and the other two pointers can be used for the integers in the I-problem. Suppose there are D dummies then the number of pointers used for all I-problems is $O(D + n/2^{4\sqrt{\log n}})$ and the time used is $O(\log n)$.

Now we have a collection C of $\leq 2^{\delta\sqrt{\log n}}$ sorted sequences of integers. We need merge all sequences in C into one sorted sequence. We first sample every $2^{\delta\sqrt{\log n}}$-th integer from each sequence $s \in C$. If a sequence has no more than $2^{\delta\sqrt{\log n}}$ integers we sample its first and last integer. We obtain a collection C' of $|C|$ sampled sequences. There are no more than $2n/2^{\delta\sqrt{\log n}}$ sampled integers. We merge every pair of sampled sequences by first making $2^{\delta\sqrt{\log n}}$ copies of each sampled sequence and merge the corresponding pair. If integer a is in sampled sequence s which is merged with every other sampled sequence, then a knows its rank in every sampled sequence. By summing these ranks a knows its rank in all sampled integers. Therefore we have sorted sampled integers into one sequence q. We then make $2^{\delta\sqrt{\log n}}$ copies of q and merge q with each sequence in C. This merge divides the original merge problem (of merging sequence in C) into $|q|+1$ submerge problems with each submerge problem to merger $|C|$ subsequences of no more than $2^{\delta\sqrt{\log n}}$ integers (one subsequence coming from one sequence in C). Since the total number of integers in each submerge problem is no more than $|C|2^{\delta\sqrt{\log n}} \leq 2^{7\sqrt{\log n}}$ we can use comparison sorting[1][10] to sort integers in each submerge problem.

Let us estimate the space complexity. Each I-problem formed at stage 0 or 1 at each level requires $O(n)$ space. The space needed for I-problems formed in later stages is geometrically decreasing. Thus $O(mn\sqrt{\log n})$ space is used for solving all G-problems. We then allocated $2^{\delta\sqrt{\log n}}$ arrays of A_i's. Thus the total space used is $O(mn\sqrt{\log n}2^{\delta\sqrt{\log n}})$. However, space can be reduced to $O(nm^{\epsilon})$ by using radix sorting.

THEOREM 4.2. *n integers in the range $\{0, 1, ..., m-1\}$ ($m \geq n$) can be sorted in $O(\log n)$ time and $O(nm^{\epsilon})$ space with $O(n\sqrt{\log n})$ operations and $O(\log(m+n))$ word length on the EREW PRAM.*

To sort n integers in the range $\{0, 1, 2, ..., m-1\}$ with word length $k\log m$ bits we modify our algorithm to sort $O(\log m/\sqrt{\log n/k})$ bits in each level and in the i-th stage we sort every $2^{2^{i+2}\sqrt{k\log n}}$ integers. This will give $O(\log n)$ time and $O(n\sqrt{\log n/k})$ operations.

THEOREM 4.3. *n integers in the range $\{0, 1, ..., m-1\}$ can be sorted in $O(\log n)$ time with $O(n\sqrt{\log n/k})$ operations and $O(k\log(m+n))$ word length on the EREW PRAM.*

5. Conclusions. We presented EREW integer sorting algorithms which outperform parallel comparison sorting. There are several open questions. An immediate one is to further improve operation complexity. Another open problem is to reduce the space complexity of the algorithm in Theorem 4.2 to linear. Applications of our algorithm to other problems will also be interesting where the advantage of parallel integer sorting over parallel comparison sorting can be made use of.

REFERENCES

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

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

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

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

[5] R. Anderson and G. Miller, *Deterministic parallel list ranking*, Algorithmica 6: 859-868(1991).

[6] P. Beame, J. Hastad, *Optimal bounds for decision problems on the CRCW PRAM*, Proc. 19th Annual ACM Symposium on Theory of Computing, 83-93(1987).

[7] V.E. Benes, *On rearrangeable three-stage connecting networks*, Bell Syst. Tech. J. , Vol. 41, 1481-1492(1962).

[8] V.E. Benes, *Mathematical Theory of Connecting Networks and Telephone Traffic*, New York: Academic, 1965.

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

[10] R. Cole, *Parallel merge sort*, SIAM J. Comput., 17(1988), pp. 770-785.

[11] S. Cook, C. Dwork, R. Reischuk, *Upper and lower time bounds for parallel random access machines without simultaneous writes*, SIAM J. Comput., Vol. 15, No. 1, 87-97(Feb. 1986).

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

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

[14] A.V. Goldberg, S.A. Plotkin, G.E. Shannon, *Parallel symmetry-breaking in sparse graphs*, SIAM J. on Discrete Math., Vol 1, No. 4, 447-471(Nov., 1988).

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

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

[17] Y. Han, *Matching partition a linked list and its optimization*, Proc. 1989 ACM Symposium on Parallel Algorithms and Architectures (SPAA'89), Santa Fe, New Mexico, 246-253(June 1989).

[18] Y. Han, *An optimal linked list prefix algorithm on a local memory computer*, Proc. 1989 Computer Science Conference (CSC'89), 278-286(Feb., 1989).

[19] 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).

[20] J. JáJá, *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

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

[22] C. P. Kruskal, L. Rudolph, M. Snir, *A complexity theory of efficient parallel algorithms*, Theoret. Comput. Sci., **71**, 95-132.

[23] T. Leighton, *Tight bounds on the complexity of parallel sorting*, IEEE Trans. Comput. C-34, 344-354(1985).

[24] S. Rajasekaran and J. Reif, *Optimal and sublogarithmic time randomized parallel sorting algorithms*, SIAM J. Comput. **18**, pp. 594-607.

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

[26] R. Raman, *The power of collision: randomized parallel algorithms for chaining and integer sorting*, Proc. 10th Conf. on Foundations of Software Technology and Theoretical Computer Science, Springer Lecture Notes in Computer Science, Vol. 472, pp. 161-175.

[27] R. E. Tarjan, U. Vishkin, *Finding biconnected components and computing tree functions in logarithmic parallel time*, Proc. 1984 Symp. Foundations of Computer Science, 12-20(1984).

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

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

[30] J. C. Wyllie, *The complexity of parallel computation*, TR 79-387, Department of Computer Science, Cornell University, Ithaca, NY, 1979.

Appendix A: Implementation of the Algorithm in Section 3. The subtlety of our algorithm is in how to do grouping, where to place dummy elements and how to maintain the space complexity within $O(n)$. Grouping should be done with linked list contraction. However, we can not apply any existing linked list contraction algorithms directly to obtain $O(\sqrt{\log n})$ time and $O(n)$ operations for a stage because we need an algorithm to do partial linked list contraction. The dummy elements generated need to be placed within $O(n)$ space so that processors can be allocated to them. For the space complexity consideration, after grouping we need sort integers within each group and this may seem requiring that we place the integers in a group in consecutive memory locations. If we allocate $O(n)$ memory for placing all integers such that all integers in a group occupy consecutive memory locations, then it would need $O(\log n)$ time while we can expend only $O(\sqrt{\log n})$ time in each stage. What we could do instead is to allocate a two dimension array with $2^{5\sqrt{\log n}}$ rows and n columns. We place the linked lists in the first row. For each group, we could put the integers in the group in the j-th column of the array if the first integer in the group is in column j. This scheme facilitates sorting. The only shortcoming of the scheme is that it uses more than $O(n)$ space. We give schemes from which all the problems mentioned above can be resolved.

For implementation purpose we reveal $\sqrt{\log n}$ bits in each stage except the last stage which reveals $10\sqrt{\log n}$ bits. A linked list is very short if it contains no more than $2^{2\sqrt{\log n}}$ integers, is short if it contains less than $2^{6\sqrt{\log n}}$ integers, is long if it contains at least $2^{6\sqrt{\log n}}$ integers. A group on a short linked list contains all integers in the list. A group on a long linked list contains at least $2^{6\sqrt{\log n}}$ but less than $2^{7\sqrt{\log n}}$ integers.

A.1. Blocked Linked List. We modify our linked list construction. Instead of linking elements(integers) from memory location to memory location, we require that every $2^{2\sqrt{\log n}}$ elements in a linked list occupy consecutive memory locations and the first element among these $2^{2\sqrt{\log n}}$ elements is at a memory cell j where $j \bmod 2^{2\sqrt{\log n}} = 0$. We call such $2^{2\sqrt{\log n}}$ elements a block. Thus if we walk down the linked list, we visit $2^{2\sqrt{\log n}}$ consecutive memory locations, then follow the pointer to another memory location, then visit another $2^{2\sqrt{\log n}}$ consecutive memory locations, and so on. We call such a linked list a blocked linked list. For all the linked lists split we maintain this property (except the linked lists split at the end of last stage). This property facilitates linked list contraction. The condition on memory cell $j \bmod 2^{2\sqrt{\log n}} = 0$ ensures that processors can be allocated to the elements in the linked lists. Because we use $n/\sqrt{\log n}$ processors, one processor is allocated for $\sqrt{\log n}$ elements or integers.

A.2. Linked List Contraction. Now consider grouping. Because linked lists are blocked, the linked list contraction for the bottom $2^{2\sqrt{\log n}}$ elements are automatically done. That is, for a linked list l_1 of length S, we can view it as being already contracted to a linked list l_2 of length $S/2^{2\sqrt{\log n}}$. For the further contraction of l_2, we can allocate one processor for each node in l_2.

Therefore we are now facing the following linked list contraction problem: Contract a linked list in $O(\sqrt{\log n})$ time and $O(n\sqrt{\log n})$ operations (note that here we can assign one processor to each node in the linked list) such that every S elements on the linked list is contracted to a single node, where $2^{c\sqrt{\log n}} \leq S \leq 2^{(c+1)\sqrt{\log n}}$

and c is a constant.

This is a very special linked list contraction problem and no known linked list contraction algorithm can solve it directly. We use the following scheme to solve this problem.

We will apply pointer jumping [30] to contract linked list. For a linked list of n nodes pointer jumping takes $O(\log n)$ time and $O(n \log n)$ operations to contract the whole list. Since we are going to contract every S elements on the linked list we should break the linked list into pieces such that each piece has S nodes of the linked list and then we apply pointer jumping on each piece. Then the pointer jumping will take $O(\log S) = O(\sqrt{\log n})$ time and $O(n \log S) = O(n\sqrt{\log n})$ operations. This scheme is a perfect one except that we do not have a scheme to break the linked list into pieces of S nodes each.

The currently best parallel algorithm [17, 18, 14] for break linked list into pieces (symmetry breaking algorithm) can break the linked list into pieces in $O(\log d)$ time and linear operations such that each piece has at least two nodes and at most $\log^{(d)} n$ nodes, where d is a constant. Here we are guaranteed that the linked list will be broken up into pieces. But we are not guaranteed that each piece contains between $2^{c\sqrt{\log n}}$ and $2^{(c+1)\sqrt{\log n}}$ nodes.

Should each broken piece contains about the same number of nodes, say T nodes, then we could apply pointer jumping on each piece for $O(\log T)$ time to contract each piece into a single node. The original linked list L_1 of n nodes is thus being contracted into a linked list L_2 of n/T nodes. Now we could apply symmetry breaking on L_2 and then pointer jumping on the pieces of the linked list after symmetry breaking. And we could repeat this symmetry breaking and pointer jumping process for $\log S/\log T$ times and would have finished linked list contraction in $O(\log S)$ time and $O(n \log S)$ operations.

The problem now is that each broken piece of the linked list can contain as few as two nodes and as more as $\log^{(c)} n$ nodes. Thus the shortest piece takes constant time to contract and the longest piece takes $O(\log^{(c+1)} n)$ time to contract when pointer jumping is applied. If we let processors working on short piece wait after they finish pointer jumping for the processors working on long piece then the progress of the algorithm will not be fast enough for us to get the $O(\sqrt{\log n})$ time for linked list contraction.

Thus the strategy we use is that when the processors working on a short piece P finish pointer jumping they check whether the neighboring pieces (the previous piece and the next piece) have finished contraction (pointer jumping). If both of its neighboring pieces have not finished contraction P will wait. If one of P's neighbor N has already finished contraction then the node P contracts into and the node N contracts into are linked on a linked list and then symmetry breaking and pointer jumping can applied to this linked list.

Since no two consecutive nodes can be in wait state at the same time, every three nodes are contracted into at most two nodes in a step. Thus the whole contraction process takes $O(\sqrt{\log n})$ time and $O(n\sqrt{\log n})$ operations.

A.3. Cope with Dummy Elements. We first show how do we handle short linked lists. The number of bits we revealed for each integer is $\leq \log n - 10\sqrt{\log n}$ before the the last stage. Thus we can have at most $n/2^{10\sqrt{\log n}}$ short linked lists before the last stage. Each short linked list can be indexed by an integer range from 0 to $n/2^{10\sqrt{\log n}}$. We can allocate an array A of size n to store only short linked lists.

$A[i2^{7\sqrt{\log n}}..(i-1)2^{7\sqrt{\log n}}-1]$ is reserved for the short linked list indexed i. Once we have a short linked list we sort it with parallel comparison sorting on the whole integer (not just the revealed bits). This entails $O(\sqrt{\log n})$ time for each short list and $O(\sqrt{\log n})$ operations for each integer and therefore it will not destroy the complexity analysis for the whole algorithm. Thereafter we only consider long linked lists.

Initially we put all n input integers into a linked list and therefore we start with a long linked list. To facilitate the later processing we add n dummy elements to the initial linked list. We put input integers and dummies alternatively in the initial linked list as $a_0, d, a_1, d, a_2, d, ..., d, a_{n-2}, d, a_{n-1}$, where d is a dummy. Therefore for every t consecutive integers on the linked list we have t dummies. These dummies will later be used for keeping the blocking property of the linked list and for representing missing patterns in a group.

Initially we have one dummy for every two consecutive elements on the linked list. We use $1/2$ to represent this ratio. After several stages this ratio will become smaller. We assume that at the current stage the ratio is $1/b$, i.e. there is a dummy in every b elements on the linked list.

After sorting integers in a group, integers with the same revealed bits (bit pattern) are consecutive on the linked list. However, the number of integers with the same revealed bits may not be a multiple of $2^{2\sqrt{\log n}}$. To maintain the blocking property of the linked list, we make use of dummy elements so that the number of integers and dummies within each group with the same revealed bit pattern become a multiple of $2^{2\sqrt{\log n}}$. For a group G of S integers in a long linked list, we split it into $2^{\sqrt{\log n}}$ linked lists. Let a linked list L split from G have T integers. We put $\lceil T/(2b)\rceil + 2\cdot 2^{2\sqrt{\log n}} - (T+\lceil T/(2b)\rceil)\%2^{2\sqrt{\log n}}$ dummies into L, where % is the modulo operation. Thus the total number of integers and dummies in L is a multiple of $2^{2\sqrt{\log n}}$. Summing over all split linked lists the total dummies we used is $< S/(2b) + 3\cdot 2^{3\sqrt{\log n}}$. Thus if $S/b \geq S/(2b) + 3\cdot 2^{3\sqrt{\log n}}$, i.e. $S/(2b) \geq 3\cdot 2^{3\sqrt{\log n}}$, then we have enough dummies. The ratio of the dummies to the elements in the split linked list is now $1/(2b)$. Thus through one stage the ratio is reduced by at most half. Since there are $\sqrt{\log n}$ stages the smallest ratio we have is $1/2^{\sqrt{\log n}}$. Because $S \geq 2^{6\sqrt{\log n}}$ we can hold $S/(2b) \geq 3\cdot 2^{3\sqrt{\log n}}$.

A.4. Sorting Each Group in Linear Space. We now show how to sort each group in linear space. For a short linked list we reveal all remaining bits of the integers on the list and then sort these integers using comparison sorting[1][10]. Since there are at most $n/2^{10\sqrt{\log n}}$ very short linked lists (at the beginning of the last stage), the total number of operations involved in sorting short linked lists is $O((n/2^{10\sqrt{\log n}})\cdot 2^{7\sqrt{\log n}}\cdot\sqrt{\log n}) \leq O(n\sqrt{\log n}/2^{3\sqrt{\log n}})$.

The sorting of integers in a group on a long linked list is done directly on integers on the linked list. For the purpose of sorting we may assume that the number of integers in a group is a power of 2. Otherwise we simply add some dummy elements to make it a power of 2 and store these dummy elements along elements of the linked list so that each memory location on the linked list stores at most two elements (this can be realized by an array of two rows). Note that we use sorting network to accomplish the sorting. In order to sort integers on a linked list, an integer on the linked list has to know the address of the integers to which it will compare with at each level of the sorting network. Because AKS sorting network has $O(\sqrt{\log n})$ levels (because we are sorting at most $2^{7\sqrt{\log n}}$ integers in the group) and because in our sorting algorithm we pack $O(\sqrt{\log n})$ integers into one word each word on the linked list has to know $O(\sqrt{\log n})$ addresses (one address for each level). Let a_i be the word at memory address $d(a_i)$ (memory address is an $O(\log n)$ bit integer). If a_i needs to compare with a_j at the first level of the sorting network, we need route address $d(a_i)$ to $d(a_j)$ and address $d(a_j)$ to $d(a_i)$. What we need here is a permutation of the memory addresses and the permutation is a fixed one (known in advance). Thus we can route the memory address through the permutation network given in Section 2. Note that butterfly network has such a "regular" structure that each node (word) on the linked list can find the address of the nodes it need to switch with at all levels of the network through pointer jumping[30]. The regularity we are talking here is that each node needs to know the nodes which is 2^i, $i = 0, 1, 2, ...$, positions away from it and this can be done by pointer jumping. AKS sorting network, on the other hand, do not have this property. Pointer jumping allow nodes on a linked list to meet with nodes which are 2^i positions (distance) away along the linked list, $i = 0, 1, 2, ...$ Let S be the number of integers in the group. Then there are $S/\sqrt{\log n}$ words. Each word has an address to be permuted and the permutation takes $O(\sqrt{\log n})$ time. Thus one permutation uses $O(S)$ operations for the group ($O(n)$ operations for all linked lists). However we need do $O(\sqrt{\log n})$ permutations for $O(\sqrt{\log n})$ addresses because AKS sorting network has that many levels. And we have to do all these permutations in $O(S)$ operations for the group ($O(n)$ operations for all linked lists). In order to solve this problem we use a modified version of our nonconservative sorting algorithm. In each node of the sorting network we compare $\sqrt{\log n}$ words with another $\sqrt{\log n}$ words in parallel instead of comparing just two words. That is, we use $\sqrt{\log n} \cdot \sqrt{\log n} = \log n$ columns in the column sort. By the blocking property of the linked list these $\sqrt{\log n}$ words occupy consecutive memory addresses. Thus for each $\sqrt{\log n}$ words we need permute only the address of the first word. When the permutation is done, the addresses of the other words can be figured out. Therefore the number of operations for each permutation is reduced to $O(S/\sqrt{\log n})$ ($O(n/\sqrt{\log n})$ for all linked lists). For all $O(\sqrt{\log n})$ permutations the total number of operations is $O(S)$ ($O(n)$ for all linked lists). In order to keep $O(\sqrt{\log n})$ time for all these $O(\sqrt{\log n})$ permutations, we do them in parallel. For the $\sqrt{\log n}$ words, we let the i-th word to participate in the i-th permutation.

The permutations performed among integers after each sort on columns can be done by routing the integers through the permutation networks given in Section 2. Again since the butterfly network has such a regular connection structure a word can find the word it will switch with through pointer jumping. We therefore showed that sorting can be done for integers on the linked list.

After sorting we need move integers to the sorted position. The problem here is that when we are sorting we are moving $\sqrt{\log n}$ bits for each integer through the AKS sorting network. In order for each integer to know the address after sorting the address which is a $\log n$ bit integer should be known to the integer. Note that such an address has $\log n$ bits and we cannot move it through the sorting network (for otherwise it will incur $O(n\sqrt{\log n})$ operations *in one stage*). We cannot move integers through sorting network either because each integer has about $\log n$ bits (it has less bits in later stages, though) instead of the $\sqrt{\log n}$ revealed bits.

Since each block has $2^{2\sqrt{\log n}}$ integers we modify the sorting within each group to first sort integers in each block and

then to sort the whole group. Sorting integers in each block can be done by

Theorem 2.4 and the relative address of the sorted position of an integer a has only $2\sqrt{\log n}$ bits. Therefore the *relative address* can be transferred back to a through AKS sorting network (as we did in the above paragraph)

and then a can use this address to index into the sorted position. After each block is sorted the integers with the same revealed bits in a block are in consecutive memory locations. Because each block has $2^{2\sqrt{\log n}}$ integers and there are only $2^{\sqrt{\log n}}$ revealed bit patters, for each bit pattern p we could use a representative integer a_p and let a_p find sorted locations (after the sort for the group) for all integers in the block with bit pattern p. Since the total number of representatives is a fraction of the total number of integers, the representatives can find the sorted positions by routing them through the sorting network.

Before the beginning of the last stage (which reveals $4\sqrt{\log n}$ bits) we use linked list ranking[5] to move all integers in a linked list into consecutive memory locations. Therefore in the last stage the integers to be sorted are based on arrays instead of linked list.