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 on parallel integer sorting. Our EREW PRAM 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 $1 \leq k \leq \log n$. When $k = \log n$ this algorithm sorts n integers in $O(\log n)$ time with linear operations. When $k = 1$ this algorithm sorts n integers in $O(\log n)$ time with $O(n\sqrt{\log n})$ operations.

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 when 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 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][12][18][20]. Fredman and Willard first showed [12] 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. We show, for the first time, that parallel integer sorting is more efficient than parallel comparison sorting on the exclusive write PRAMs.

The parallel computation model we use is the EREW PRAM model[19]. Parallel algorithms can be measured by their time complexity and operation complexity which is the time processor product. The operation complexity of a parallel algorithm can also be compared with the time complexity of the best sequential algorithm for the same problem. Let T_1 be the time complexity of the best sequential algorithm for a problem, T_p be the time complexity of a parallel algorithm using p processors for the same problem. Then $T_p \cdot p \geq T_1$. That is, T_1 is a lower bound for the operation complexity of any parallel algorithm for the problem. A parallel algorithm is said to be opitmal if its operation complexity matches the time complexity of the best sequential algorithm, i.e. $T_p \cdot p = O(T_1)$.

In order to outperform parallel comparison sorting on the exclusive 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 and outperforms the operation lower bound for parallel comparison sorting. 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 $\Omega(\log n)$[10]. The operation lower bound for parallel comparsion sorting is $\Omega(n \log n)$ due to the time lower bound for sequential comparison sorting. Known parallel algorithms failed to outperform the lower bound for parallel comparison sorting because of the following reasons.

1. Parallel algorithms are known [2][4][11][18][24] to have operation complexity of $o(n \log n)$ when they are running slower than the time lower bound for parallel comparison sorting. But they failed to have $o(n \log n)$ operations when made to run at 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 $\Omega(\log n)$[10]. It is not clear how to make the algorithm in [2] to run in $O(\log n)$ time.

*Electronic Data Systems, Inc., 750 Tower Drive, CPS, Mail Stop 7121, Troy, MI 48098. yhan01@cps.plnin.gmeds.com, http://welcome.to/yijiehan

†Computer Science Telecommunications Program, University of Missouri — Kansas City, 5100 Rockhill Road Kansas City, MO 64110. xshen@cstp.umkc.edu

Also the CRCW algorithm in [4][18] 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][8][26] that have operation complexity $o(n \log n)$ and time which matches the time lower bound for parallel comparison sorting when sorting on small integers. These results fail to outperform parallel comparison sorting when sorting on large integers. For example, the previous best results in [2][11] showed that n integers in the range $\{0, 1, ..., 2^{O(\sqrt{\log n})}\}$ can be sorted in $O(\log n)$ time and linear operations. No previous deterministic algorithms showed that 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][15] 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) we improve 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 results demonstrate the curve for operation complexity $O(n\sqrt{\frac{\log n}{k}})$ when using word length $k \log(m+n)$, where $1 \leq k \leq \log n$, while our algorithm runs in $O(\log n)$ time. When $k = 1$ our algorithm uses standard word length $\log(m+n)$ and runs in $O(\log n)$ time with $O(n\sqrt{\log n})$ operations. This algorithm outperforms parallel comparison sorting on the CREW and EREW PRAMs. Note also that the integer sorting algorithms presented in this paper are stable sorting algorithms.

There are many previous results on parallel integer sorting [2][4][8][11][14][15][18][21][23][24][25][26]. 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[20] 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[18] showed conservative integer sorting algorithms which sorts 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][8][14][18] 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. In fact, for sorting n integers in the range $\{0, 1, ..., n-1\}$ which is considered to be the most important and standard case, previous best conservative parallel algorithms running in $O(\log n)$ time on CREW and EREW PRAM use $O(n \log n)$ operations. Rajasekaran and Sen[24], Albers and Hagerup[2] and Dessmark and Lingas[11] were able to reduce the number of operations to $o(n \log n)$ on the CREW PRAM and EREW PRAM when the running time is enlarged to 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][24] sort in $O(\log n \log\log n)$ time with $O(n \log n / \log\log n)$ operations. Very recently Dessmark and Lingas presented an improved EREW algorithm[11] which sorts in $O(\log^{3/2} n)$ 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][9] (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. 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[11] while our algorithm runs faster (in $O(\log n)$ time) than their algorithms (in $O(\log n \log\log n)$ time on the CREW PRAM and in $O(\log^{3/2} n)$ time on the EREW PRAM).

For the interger sorting problem of sorting n interger in the range $\{0, 1, 2, ..., m-1\}$, all previous EREW and CREW conservative algorithms[2][11][21][24][26] require $O(n \log n)$ operations when m is large, even when the time complexity is enlarged to polylogarithmic of n. Actually the number of opearations of best previous results is larger than $O(n \log n)$, however, we could as-

sume 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.

Note that the best sequential integer sorting algorithm using linear space is due to Andersson[3]. Its worst case running time is $O(n\sqrt{\log n})$. We present an algorithm (Theorem 4.1.) with $O(\log^{3/2} n)$ time and $O(n\sqrt{\log n})$ operations and it runs in linear space. Therefore the operation complexity of our parallel algorithm matches the worst case time complexity of Andersson's sequential algorithm.

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[15] showed that if $w = O(n \log n \log m)$ the sorting can be done in $O(n)$ sequential time or in $O(\log n)$ time on a EREW PRAM with $O(n/\log n)$ processors. 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 $O(\log^2 n)$ time with $O(n)$ operations on the EREW PRAM. The result of Andersson et al.[4] show that the sorting can be done in $O(\log n)$ time with $O(n)$ operations on the EREW PRAM with a word length of $O((\log n)^{2+\epsilon} \log m)$. Dessmark and Lingas showed that the sorting can be done 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 $O(\log n)$ 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 $O(\log n)$ time with $O(n)$ operations. 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.

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. This EREW algorithm is based on the AKS sorting network[1], Leighton's column sort[22] and Benes permutation network[7].

Since the word length is $O(\log n \log m)$ we can store $\log n$ integers in a word. Using the test bit technique[2][4] we can do pair-wise comparison of the $\log n$ integers in a word with the $\log n$ integers in another word in constant time using one processor. Moreover, using the result of the comparison the $\log n$ larger integers in all pairs can be extracted into one word and the $\log n$ smaller intergers in all pairs can be extracted into another word and this can also be done in constant time using one processor[2][4]. Without loss of generality we may also assume that $\log n$ is a power of 2. We first pack n input integers into $n/\log n$ words with each word containing $\log n$ integers. We then imagine an AKS sorting network being built on these $n/\log n$ words. On the AKS sorting network we compare two words at each internal node of the network. 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 $\log n$ sets with the i-th set containing i-th integers in all $n/\log n$ words. In terms of Leighton's column sort[22] we can view that we place n integers in $\log n$ columns with each column containing $n/\log n$ integers. At the output of the AKS sorting network, every column is sorted. The principle of Leighton's column sort says that to sort n integers we need only sort $\log n$ columns for a constant number of times and perform a fixed permutation among the n integers between each sort (of columns). Besides these fixed permutations are simple permutations such as shuffle, unshuffle and shift. Applying this principle, we perform a fixed permutation among the n integers after they are output from the AKS sorting network. The permutation can be done by disassembling the integers from the words, applying the permutation and then reassembling the integers into words. Thus each sorting on columns and permutation can be done in $O(\log n)$ time. According to Leighton's column sort we need only a constant number of passes of sorting and permutation. Thus the sorting of n integers can be done in $O(\log n)$ time. The operations consumed by our algorithm is $O(n)$. Note also that the sorting can be made stable by appending address bits to each integer.

For our purpose 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. 1.). For permutations N can be used to emulate the Benes permutation network[7]. Each stage of the butterfly network emulates the processor connection along a dimension on the hypercube and switches integers between words or within words (this is where we need $\log n$ to be a power of 2). Therefore each stage of the but-

terfly network can be done in constant time. Because butterfly network has $O(\log n)$ stages, the permutaion 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.

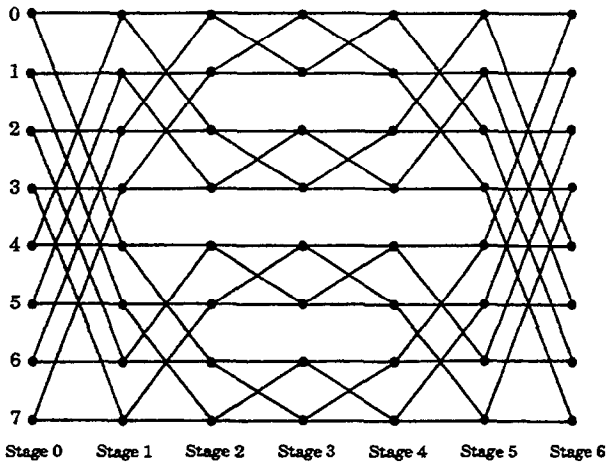

Figure 1: A permutation network.

THEOREM 2.1. *n intergers 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.*

The principle of Theorem 2.1 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[22] to sort n integers in constant number of passes (of sorting columns and permuting), 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 2.1. Also we may use more 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 $\log n$.

The following corollary can now be easily shown.

COROLLARY 2.1. *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.*

3 Sorting Integers in $\{0, 1, ..., n-1\}$

We first consider sorting with word length $O(\log n)$. For our purpose we assume that $\sqrt{\log n}$ is a power of 2.

3.1 Outline

If input integers with the same value are linked in a linked list according to 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 $\{0, 1, 2, ..., n-1\}$, the first integer in each linked list can put themselves into buckets. Because there are only n buckets integers dropped into the buckets can be collected in $O(\log n)$ time and $O(n)$ operations.

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 intgers 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 very short if it contains no more than $\log n$ elements(integers), 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 shall discuss further how to do this grouping later. For the moment 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. We then sort integers in each group in parallel. Because revealed bits for the previous stages for integers in the linked list is 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 the column sort cannot be applied (see the paragraph immediately after Theorem 2.1). If a linked list is very short we simply sort the revealed bits on the list by a comparison sorting algorithm[1][9]. Since there are at most $n/2^{\sqrt{\log n}}$ very short linked lists at the beginning of the last stage (this is where we reallocate processors for the last time) we can allocate one processor for each integer in such linked lists according to revealed bit patterns of each such linked list. At the beginning of other stages there are much less than $n/2^{\sqrt{\log n}}$ very short linked lists.

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 existense 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.

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.

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.

3.2 Implementation

The subtlety of our algorithm is at 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 $4\sqrt{\log n}$ 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.

We modify our linked list construction. Instead of linking elements(integers) from memory location to memory location, we require that every $2^{2\sqrt{\log n}}$ ele-

ments 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.

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. We then repeatedly apply symmetry breaking schemes by Han[16][17] and Beame[13] to break l_2 into lists of length no more than $\log^{(c)} n$ (c is a constant, $\log^{(1)} n = \log n$, $\log^{(c)} n = \log\log^{(c-1)} n$) and pointer jumping technique of Wyllie[27] to contract l_1 until at least S elements are contracted into one node, where for a short linked list S is the number of elements on the linked list and for a long linked list S is at lease $2^{6\sqrt{\log n}}$ but no more than $2^{7\sqrt{\log n}}$. The contraction can thus be done in $O(\sqrt{\log n})$ time with $O(S)$ operations for the linked list ($O(n)$ operations for all linked lists).

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 add minimum number of dummy elements for each bit pattern so that the number of integers within each group with the same bit pattern become a mutiple of $2^{2\sqrt{\log n}}$. For a short linked list, if the number of different bit patterns for the integers in the list is B we add at most $B2^{2\sqrt{\log n}}$ dummy elements. There are at most $n/2^{4\sqrt{\log n}}$ different bit patterns (this is the maximum number of different bit patters at the beginning of the last stage and we do not maintain blocking property at the end of the last stage). Therefore the totol number of dummy elements added for all short linked lists for all stages is bounded by $n/2^{2\sqrt{\log n}}$. For a group in a long linked list, we add at most $2^{3\sqrt{\log n}}$ dummy elements. Note that, if a bit pattern does not exists among the revealed bits for the integers in the group, $2^{2\sqrt{\log n}}$ dummy elements will be added to keep the blocking property. Since each group has at least $2^{6\sqrt{\log n}}$ integers the total number of dummy elements added for all long linked lists for all stages is bounded by $n\sqrt{\log n}/2^{3\sqrt{\log n}}$.

As dummy elements are created, where should them be stored? We need a scheme such that all input integers as well as dummy elements created are stored in $O(n)$ space so that processors can be allocated to them. We use an array of 3 rows and n columns. The input integers and their links are stored in the first row. A block of $2^{2\sqrt{\log n}}$ elements (containing dummy elements) for integers on each short linked list is stored at the second row. There is one such block for each revealed bit pattern. Therefore memory at the second row can be allocated according to the revealed bits. The third row is used for storing some integers in long linked lists. In each stage we sort integers in a group on a long linked list. Let us assume this sorting is done. Let S_i be the number of integers (within the group) with the same revealed bit pattern p_i (these integers are now consecutive on the linked list). For every $2^{4\sqrt{\log n}}$ integers along the linked list among these S_i integers we move the last $2^{2\sqrt{\log n}}$ integers from the first row to the third row, storing them in columns where the first $2^{2\sqrt{\log n}}$ integers are stored in the first row. This placement ensures that there is at most $2^{2\sqrt{\log n}}$ integers stored on the third row for every $2^{4\sqrt{\log n}}$ integers along the linked list at the first row. We call the $2^{4\sqrt{\log n}}$ integers with the same revealed bit pattern along the linked list at the first row a segment. What we did is essentially place a block on the third row for each segment on the first row. Because the group has at least $2^{6\sqrt{\log n}}$ integers and only $2^{\sqrt{\log n}}$ revealed bit patterns, roughly speaking there are at least $(2^{6\sqrt{\log n}} - 2^{5\sqrt{\log n}})/2^{4\sqrt{\log n}}$ segments. Thus we have placed at least $2^{3\sqrt{\log n}}$ integers on the third row. Therefore we have at least $2^{3\sqrt{\log n}}$ empty positions ($2^{\sqrt{\log n}}$blocks) at the first row where we can store newly created dummy elements. After linked list in a group is split and joined with linked lists in the neighboring groups, there are at most $2^{2\sqrt{\log n}}$ integers stored in the third row for $2^{4\sqrt{\log n}}$ consecutive integers on a linked list in the first row. For the next stage, there will be integers stored in the third row which is carried over

from previous stages and there will be new integers to be stored in the third row. In each stage we first "remove" integers from the third row (and may place them in the first row after sorting the integers in the group). After sorting within the group we place new integers on the third row. Suppose there were at most $t2^{2\sqrt{\log n}}$ integers on the third row for every $2^{4\sqrt{\log n}}$ integers (a segment) on a linked list at the first row. Let S be the number of integers within the group of a long linked list. Then there were at most $St2^{2\sqrt{\log n}}/2^{4\sqrt{\log n}}$ integers at the third row. Roughly speaking there are at lease $(S - 2^{5\sqrt{\log n}})/2^{4\sqrt{\log n}} < S/(c2^{4\sqrt{\log n}})$ seqments, where $c < 1.5$ is a constant. Distribute these integers among these segments, there will be $ct2^{2\sqrt{\log n}}$ integers per segment. Plus we will add $2^{2\sqrt{\log n}}$ integer to the third row for each segement in the current stage, the number of integers stored at the third row per segement will be $< 2t2^{2\sqrt{\log n}}$. These $2t2^{2\sqrt{\log n}}$ integers have the same revealed bit pattern as that of the $2^{4\sqrt{\log n}}$ integers stored in the first row (note that integers stored in the third row in the previous stage are "removed" and in the current stage they may be placed in the first row (or third row)). After a total of $\sqrt{\log n} - 4$ stages, there are at most $2^{3\sqrt{\log n}}$ integers stored in the third row for every segment on the linked list at the first row. The scheme given here ensures that all integers and dummy elements are stored in $O(n)$ space.

Due to page limit we omit the presentation on how to sort each group in linear space.

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.

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][11][24], even allowing polylogarithmic running time, will eventually use $O(n \log n)$ operations when m is sufficiently large. What we have achieved is an algorithm which sorts in $O(\log n)$ time with $O(n\sqrt{\log n})$ operation. The presentation of this algorithm will take too much space. However, this algorithm is a speeded-up version of a slower but simpler algorithm which captures the main ideas of the fast algorithm. Here we present the details of this simpler algorithm. This algorithm runs in $O(\log^{3/2} n)$ time 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 2.1). 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 several paragraphs.

In our algorithm the computation is organized into $\sqrt{\log n}$ levels. Each level represents a pass explained in the above paragraph. 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. There are a total of $2\sqrt{\log n}$ stages in the algorithm.

The computation at level i, $0 \leq i < \sqrt{\log n}$, 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 describe the computation at levels 0 and 1 and then generalize it to arbitrary levels.

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. 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 $n/2^{s\sqrt{\log n}}$ sorted sequences. 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 $a^{[\sqrt{\log n}-1]} = b^{[\sqrt{\log n}-1]}$ (note that all these integers are now consecutive in memory). Once this condition is not satisfied a will be moved to level 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, 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 a 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][9]. 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 $0 \leq i < n/2^{\sqrt{\log n}}$. If there are integers a in $I[i2^{\sqrt{\log n}} : (i+1)2^{\sqrt{\log n}} - 1]$ 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 in stage 0 of level 0 (one group 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). 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. 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, for a group g in C which are passed down at stage 1 of level 0 the integers a in g are not sorted by $a^{[\sqrt{\log n}-2]}$ and therefore it cannot participate in the $2^{\sqrt{\log n}}$-way merge directly. What we do is to first sort integers a in g by bit $[\sqrt{\log n} - 2]$ and then perform the merge. Becuase g contains less than $2^{\sqrt{\log n}}$ 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 2.1. The situation where g contains too few integers can also be treated.

Thus at level 1 we are forming sorted sequences (sorted by bits $[\sqrt{\log n} - 2 : \sqrt{\log n} - 1]$) and repeatedly merge the sorted sequences. 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 $2\sqrt{\log n}$ 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 pointing to the first integer in p and using another pointer from the last integer in p to pointing 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 integer sorted.

At the end of each stage of our algorithm we use linked list ranking[5] and standard parallel prefix computation[19] 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 $x \leq 2^{\sqrt{\log n}}$-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 has 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 2.1. Otherwise $N \geq 2^{2\sqrt{\log n}}$ and we sample every $2^{\sqrt{\log n}}$-th integer from each of the $x \leq 2^{\sqrt{\log n}}$ 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][9]. We make x copies of S. We then merge one copy of S with one sequence in C. Suppose s_1, s_2, $s_1 \leq s_2$, are two consecutive integers in S. Then there are no more than $2^{\sqrt{\log n}}$ integers in each sequence in C which are $\leq s_2$ and $\geq s_1$ (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 2.1 to sort all integers in the subproblem. Again the situation where a merging subproblem contains too few integers can be treated.

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.*

The presentation of the algorithm for the following theorem takes several pages and will be given in the full version of the paper.

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

Note that the algorithm of Theorem 4.2 does use nonlinear space. This is a disadvantage of our algorithm which we currently do not know how to overcome.

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 $O(\log m \cdot \sqrt{k/\log n})$ bits in each level and use $2^{\sqrt{k \log n}}$-way merge at each level. This will give $O(\log n)$ time and $O(n\sqrt{\frac{\log n}{k}})$ operations.

References

[1] M. Ajtia, J. Komlós, E. Szemerédi, *Sorting in $c \log n$ parallel steps*, Combinatorica, **3**, 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, *Mathematical Theory of Connecting Networks and Telephone Traffic*, New York: Academic, 1965.

[8] 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).

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

[10] 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).

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

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

[13] 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).

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

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

[16] 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).

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

[18] 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).

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

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

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

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

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

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

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

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

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