More Efficient Parallel Integer Sorting

Yijie Han[1] and Xin He[2]

[1] School of Computing and Engineering
University of Missouri at Kansas City
Kansas City, MO 64110, USA
hanyij@umkc.edu,

[2] Department of Computer Science and Engineering
University at Buffalo, The State University of New York
201 Bell Hall
Buffalo, NY 14260-2000, USA
xinhe@buffalo.edu

Abstract. We present a more efficient CREW PRAM algorithm for integer sorting. This algorithm sorts n integers in $\{0, 1, 2, ..., n^{1/2}\}$ in $O((\log n)^{3/2}/\log\log n)$ time and $O(n(\log n/\log\log n)^{1/2})$ operations. It also sorts n integers in $\{0, 1, 2, ..., n-1\}$ in $O((\log n)^{3/2}/\log\log n)$ time and $O(n(\log n/\log\log n)^{1/2}\log\log\log n)$ operations. Previous best algorithm [13] on both cases has time complexity $O(\log n)$ but operation complexity $O(n(\log n)^{1/2})$.

Keywords: Algorithms, design of algorithms, bucket sorting, integer sorting, PRAM algorithms.

1 Introduction

Sorting is a classical problem which has been studied by many researchers [1][2][3][6][11][12][13][14][16][17][18][19]. For elements in an ordered set comparison sorting can be used to sort the elements. 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. In this paper we study parallel integer sorting and present an algorithm which outperforms the operation complexity of the best previous result.

The parallel computation model we use is the PRAM model[15] which is used widely by parallel algorithm designers. Usually three variants of PRAM models are used in the design of parallel algorithms, namely the EREW (Exclusive Read Exclusive Write) PRAM, the CREW (Concurrent Read Exclusive Write) PRAM and the CRCW (Concurrent Read Concurrent Write) PRAM[15]. 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. 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 optimal if its operation complexity matches the time complexity of the best sequential algorithm, i.e. $T_p \cdot p = O(T_1)$.

On the CREW PRAM the best previous integer sorting algorithm [13] sorts n integers in $O(\log n)$ time and $O(n(\log n)^{1/2})$ operations. In this paper we study the problem of sorting n integers in $\{0, 1, ..., n^{1/2}\}$ and in $\{0, 1, ..., n-1\}$. The best previous result for these two cases due to Han and Shen [13] also sorts in $O(\log n)$ time and $O(n(\log n)^{1/2})$ operations. In this paper we present a CREW PRAM algorithm which sorts n integers in $\{0, 1, ..., n^{1/2}\}$ in $O((\log n)^{3/2}/\log\log n)$ time and $O(n(\log n/\log\log n)^{1/2})$ operations. It also sorts n integers in $\{0, 1, ..., n-1\}$ in $O((\log n)^{3/2}\log\log n)$ time and $O(n(\log n/\log\log n)^{1/2}\log\log\log n)$ operations.

When randomization is used usually better or even optimal algorithms can be achieved. Rajasekaran and Reif first achieved an optimal randomized parallel sorting algorithm [18]. Reif and Valiant first achieved an optimal randomized parallel network sorting algorithm [19].

Parallel integer sorting is such a fundamental problem in parallel algorithm design and many renowned researchers worked on this problem relentlessly. The milestones on parallel integer sorting on exclusive write PRAMs include 1997 Albers and Hagerup's paper [2] published on Information and Computation and 2002 Han and Shen's improvement [13] published on SIAM Journal on Computing. There are many results of many researchers published before Albers and Hagerup's work without much progress passing over the $O(n \log m)$ operations for sorting n integers in $\{0, 1, .., m-1\}$. After Han and Shen's work there is virtually no progress ever since. We worked very hard and only achieved the not so big improvements presented in this paper. To our experience significant improvement over Han and Shen's work [13] on the operation complexity for parallel integer sorting is very difficult. So to speak that the results we have achieved and presented here is significant.

2 Nonconservative Sorting

First we will show the EREW PRAM algorithm in [13] to sort $n_1 = 2^{4(\log n)^{1/2}}$ integers in $\{0, 1, ..., 2^{(\log n)^{1/2}}\}$ with word length (the number of bits in a word) $\log n$. This algorithm is based on the AKS sorting network[1], Leighton's column sort[16], Albers and Hagerup's test bit technique[2] and the Benes permutation network[4][5].

Because the word length is $O(\log n)$ we can store $c(\log n)^{1/2}$ integers in a word for a small constant c. Using the test bit technique[2][3] we can do pairwise comparison of the $c(\log n)^{1/2}$ integers in a word with the $c(\log n)^{1/2}$ integers in another word in constant time using one processor. Moreover, using the result of the comparison the $c(\log n)^{1/2}$ larger integers in all pairs in the two words under comparison can be extracted into one word and the $c(\log n)^{1/2}$ smaller integers in all pairs in these two words can be extracted into another word and this can also be done in constant time using one processor[2][3]. Without loss of generality we may also assume that $c(\log n)^{1/2}$ is a power of 2. We first pack n_1 input integers into $n_2 = n_1/(c(\log n)^{1/2})$ words with each word containing $c(\log n)^{1/2}$ integers. We then imagine an AKS sorting network [1] being built on these n_2 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 $c(\log n)^{1/2}$ integers in one word with the $c(\log n)^{1/2}$ integers in another word in parallel. At the output of the AKS sorting network we have sorted $c(\log n)^{1/2}$ sets with the i-th set containing i-th integers in all n_2 words. In terms of Leighton's column sort[16] we can view that we place n_1 integers in $c(\log n)^{1/2}$ columns with each column containing n_2 integers. The i-th column, $0 \leq i < c(\log n)^{1/2}$, contains the i-th integer of every word. At the output of the AKS sorting network, every column is sorted. The principle of Leighton's column sort says that to sort n_1 integers we need only to sort all $c(\log n)^{1/2}$ columns independently and concurrently for a constant number of times (passes) and perform a fixed permutation among the n_1 integers after each pass. Besides, these fixed permutations are simple permutations such as shuffle, unshuffle and shift. Applying the column sort principle, we perform a fixed permutation among the n_1 integers when they are output from the AKS sorting network after each pass. The permutation can be done by disassembling the integers from the words, applying the permutation and then reassembling the integers into words. Thus each pass consisting of sorting on columns and then permutation can be done in $O((\log n)^{1/2})$ time and $O(n_1)$ operations. According to Leighton's column sort we need only a constant number of passes in order to have all the n_1 integers sorted. Thus the sorting of n_1 integers can be done in $O((\log n)^{1/2})$ time and $O(n_1)$ operations.

For our purpose (see later section that we have integers not in an array but in a linked list) we also need the following scheme to accomplish the permutation mentioned above. The permutation should 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.). Network N can be used to emulate the Benes

permutation network[4][5] to perform permutations. Each stage of the butterfly network emulates the processor connection along a dimension on the hypercube and switches integers between words or within words (within words means each integer is switched with another integer in the same word. This is where we need $c(\log n)^{1/2}$ to be a power of 2). Because $c(\log n)^{1/2}$ is a power of 2 each stage of the butterfly network can be done in constant time even when integers are switched within words. Because butterfly network has $O((\log n)^{1/2})$ stages, the permutation can be done in $O((\log n)^{1/2})$ time. Because there are only n_2 words the operation complexity is time×processors= $O((\log n)^{1/2}) \times n_2 = O(n_1)$. Note that since the permutations we performed here are fixed permutations according to Leighton's column sort, the setting of the switches in the butterfly network can be precomputed (according to the way Benes permutation network is used to perform permutations).

The following Lemmas 1 and 2 are the cornerstone of the paper [13] on SIAM Journal on Computing.

Sorting integers into linked lists means, after sorting, integers of the same value are in the same linked list and integers of different values are in different linked lists. It does not imply integers of the same value are packed into consecutive locations.

Lemma 1 [13]: n integers in the range $\{0, 1, ..., 2^{(\log n)^{1/2}}\}$ can be sorted into linked lists on the EREW PRAM with word length $O(\log n)$ in $O((\log n)^{1/2})$ time using $O(n)$ operations and $O(n)$ space. □

Lemma 2 [13]: n' integers in $\{0, 1, ..., 2^{t(\log n)^{1/2}}\}$ can be sorted into linked lists on the EREW PRAM with word length $\log n$ in $O(t(\log n)^{1/2})$ time and $O(tn')$ operations. □

Here in Lemma 2 n' is not related to n. Lemma 2 is essentially the t iterations of execution of Lemma 1.

Note that the result of Cook et al. [7] says that if we sort these integers in an array it will need $\Omega(\log n)$ time. The property of sorting into linked lists and the small range of values for integers enabled Lemmas 1 and 2 to be proved in [13].

3 Sorting n Integers in $\{0, 1, ..., 2^{c(\log n \log\log n)^{1/2}}\}$

We consider the problem of sorting n integers in the range $\{0, 1, ..., 2^{c(\log n \log\log n)^{1/2}}\}$ on the CREW PRAM with word length $O(\log n)$, where c is a small constant. For our purpose we assume that $(\log n/\log\log n)^{1/2}$ is a power of 2

In the first stage we pack every $(\log n/\log\log n)^{1/2}$ integer into a word (called original word later). This results in a set S_1 of $n_3 = n/(\log n/\log\log n)^{1/2}$ words. We now show how to sort these n_3 words in S_1.

The first step of this stage is to sort the integers (each having $c(\log n \log\log n)^{1/2}$ bits) within each word. This is done by a table lookup because we can precompute such a table of size n^c. This takes constant time (here we used concurrent read).

Then we take the most significant $(\log\log n)/4$ bits from each integer in each word and pack them together to obtain a word containing $(\log n/\log\log n)^{1/2}(\log\log n)/4$ bits. We first use a mask to extract these bits as shown in the first step in Fig. 2 (Applying mask). We cannot pack these extracted bits in a word together independently for each word because of complexity considerations. Therefore we shift the bits in a word and then do bitwise OR with another word to combined two words into one word, and we repeatedly do this (repeat $\log\log\log n$ times) to combine $\log\log n$ words into one word. This is step 2 in Fig. 2 (Shift and bitwise OR) and takes $O(\log\log\log n)$ time and $O(n_3)$ operations. Now all the extracted bits are stores in $n_3/\log\log n$ words. Within each words there are null bits between two blocks of extracted bits and therefore we pack extracted bits to let them occupy consecutive bits in a word. We do this independently for each word and because there are $n_3/\log\log n$ words we can afford this. This is the step 3 in Fig. 2 (Compack). This step takes $O(\log\log n)$ time (Because there are $(\log n/\log\log n)^{1/2}$ blocks of extracted bits in one word. Using constant operations we can reduce the number of blocks in a word w by half by taking half of the blocks in w out and put them in another word w_1 then shift bits in w_1 and then do $w\,OR\,w_1$.) and $O(n_3/\log\log n \times \log\log n) = O(n_3)$ operations. Now although extracted bits are packed, the order they appear in a word is *extracted bits from original word1; extracted bits from original word2; ...; extracted bits from original word(*$\log\log n$*); extracted bits from original word1; extracted bits from original word2; ...; extracted bits from original word(*$\log\log n$*);...*. Extracted bits come from different original words because of step 2 in Fig. 2. Our objective is to pack extracted bits in each original word and store them in one word. Therefore we now do step 4 in Fig 2. (Applying mask) and in $\log\log\log n$ steps and $O(n_3)$ operations we separate (disassemble) one word into $\log\log n$ words and extracted bits from each original word is now in an independent word. Because of step 3 in Fig. 2 the extracted bits are somewhat compacked in a word and therefore we can again combine words together. This times we can let extracted bits from one original word being consecutive but not compacked. This is step 5 in Fig. 2 (Shift and then bitwise OR). This step takes $O(\log\log\log n)$ time and $O(n_3)$ operations. Now again we have put all extracted bits in $n_3/\log\log n$ words. And now we do step 6 in Fig. 2 (compack) independently for each word. The complexity of this step is similar to that of step 3 (but now we have $(\log n/\log\log n)^{1/2}\log\log n$ blocks) and takes $O(\log\log n)$ time and $O(n_3)$ operations. Now we have extracted bits from each original word packed in consecutive bits of a word. Now we do step 7 in Fig. 2, i.e. separate extracted bits from each original word into an independent word. This step is similar to step 4 and takes $O(\log\log n)$ time and $O(n_3)$ operations.

Thus it takes $O(\log\log n)$ time and $O(n_3)$ operations for all the steps in Fig. 2. We call the set of these words obtained at the end of Fig. 2 S_2. Note that because many extracted bits in an original word have the same value (there are more integers in a word ($(\log n/\log\log n)^{1/2}$ of them) than the number of different values of extracted bits ($2^{(\log\log n)/4}$ of them) and integers within an original word has been sorted, therefore a word in S_2 of $(\log n/\log\log n)^{1/2}(\log\log n)/4$

bit can have only $\sum_{i=1}^{2^{(\log\log n)/4}-1} \binom{(\log n/\log\log n)^{1/2}}{i} < (\log n)^{1/2}$ values (different sorted situation corresponds to different ways of setting the position of first integer (extracted bits) among the integers (extracted bits') of the same value (except the first integer which assumes position 0)). Thus a word in S_2 can be uniquely represented by an integer i within 0 and $2^{(\log n)^{1/2}}-1$. Therefore i can be represented using no more than $(\log n)^{1/2}$ bits. Again we can use table lookup to convert a $(\log n/\log\log n)^{1/2}(\log\log n)/4$ bit integer in S_2 to an integer of $(\log n)^{1/2}$ bits. We let set S_3 to be the set of $(\log n)^{1/2}$-bit integers converted from integers in S_2. Each word in S_3 corresponds to a word in S_1.

We now partition the n_3 words of S_3 into $n_3/2^{4(\log n)^{1/2}}$ groups with each group containing $2^{4(\log n)^{1/2}}$ words. We then sort every group concurrently using the algorithm in Section 2. We spend $O((\log n)^{1/2})$ time and $O(n_3)$ operations.

We may assume that every integer value in $\{0, 1, ..., 2^{(\log n)^{1/2}} - 1\}$ (for a word) exists in each group. If such an integer value does not exist within a group we add a dummy word to the group to represent this integer value. We thus added no more than $2^{(\log n)^{1/2}}$ dummy words to each group which account for a very small fraction of the total number of words in the group. Now because words in each group has been sorted we can make $2^{(\log n)^{1/2}}$ linked lists for each group with each linked list linking all integers with the same integer value in the group together. Then we join a linked list for integer value i in a group g with lined lists for integer value i of g's left and right neighboring groups. With the help of dummies we thus obtained $2^{(\log n)^{1/2}}$ linked lists for all groups.

Now we can link words in S_1 the same way as we link words in S_3 because each word in S_1 corresponds to a word in S_3. The time complexity is $O((\log n)^{1/2})$ and the operation complexity is $O(n_3)$.

This accomplishes the first stage.

In each subsequent stage we take the next $(\log\log n)/4$ bits from each integer in a word in S_1 to form a word in set S_2 (now there is a set S_2 for each linked list). Then from S_2 we obtain S_3 (again one for each linked list) and then we sort each group of S_3 of each linked list.

Now we discuss how each linked list is split in each stage. Elements in each linked list are sorted (using the sorting algorithm in Section 2 and here we need to do permutation in the sorting algorithm using the butterfly network, see also [13]) in each stage and this linked list is going to be split into multiple linked lists such that elements of the same value will be in the same linked list and elements of different value are sorted into different linked lists.

A linked list is short if it contains less than $2^{4(\log n)^{1/2}}$ elements (words), is long if it contains at least $2^{4(\log n)^{1/2}}$ elements. We first group every consecutive S elements in a 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 $2^{4(\log n)^{1/2}}$ and no more than $2^{5(\log n)^{1/2}}$.

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 $t \leq 2^{(\log n)^{1/2}}$ linked lists such that each linked list split contains all words whose integer values are the same, where

t is the number of different integer values. Here we note that for short linked list t could be less than $2^{(\log n)^{1/2}}$ (for example if all integer values are the same t will be equal to 1).

If the linked list is long we will always split the linked list into exactly $2^{(\log n)^{1/2}}$ linked lists no matter how many different integer values are there. After sorting in each group, words in each group are split into $2^{(\log n)^{1/2}}$ linked lists. If an integer value among the $2^{(\log n)^{1/2}}$ values does not exist we create a linked list containing only one dummy element representing this integer value. Again as we stated above, no more than $2^{(\log n)^{1/2}}$ 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 integer values are joined together. With the help of those dummy elements we now have split a long linked list into exactly $2^{(\log n)^{1/2}}$ 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.

Since each group on a long linked list has at least $2^{4(\log n)^{1/2}}$ elements and since each such a group creates at most $2^{(\log n)^{1/2}}$ dummy elements, the total number of dummy elements created in a stage is at most $n_3/2^{3(\log n)^{1/2}}$. 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 $O((\log n)^{1/2})$ stages the total number of dummy elements generated is no more than $O(n_3(\log n)^{1/2}/2^{3(\log n)^{1/2}})$.

Because integers are now on linked lists, linked list contraction is needed to form groups. This paragraph describes linked list contraction and is somewhat involved. Readers who are not very familiar with symmetry breaking and linked list contraction can skip this paragraph. We apply symmetry breaking schemes by Han[9][10] and Beame[8] to break a linked list into sublists of length no more than $\log^{(c)} n$ in $O(\log c)$ time for a constant c. Pointer jumping[20] is then executed for each sublist. When pointer jumping finishes the sublist is contracted into one node. Since the length of these sublists are different some sublists finish pointer jumping faster and some sublists finish pointer jumping slower. If a sublist is contracted into a single node v, the processor associated with v checks to see if the neighboring sublists also have been contracted into single nodes. If one of its neighboring sublist is contracted into a single node then nodes representing the sublists form a new list and symmetry breaking and pointer jumping can be applied to this new list. And therefore the contraction process continues. If v finds out that both of its neighboring sublist have not finished pointer jumping then v becomes inactive. In this case v will be picked up (activated and contracted together with) by the contracted node representing the neighboring sublist which first finishes pointer jumping. We define one step for a node as first picking up its inactive neighbors and then if it is still active performing symmetry breaking and a pointer jump. This whole contraction process can be viewed as contracting a linked list of length l to a linked list of length $2l/3$ in a step because if a node

is inactive then both of its neighbors are active in the contraction process. Thus to contract S elements into a node takes only $O(\log S)$ time. For a long linked list each group can be kept between $2^{6(\log n)^{1/2}}$ and $2^{7(\log n)^{1/2}}$. Thus for each stage the contraction can thus be done in $O((\log n)^{1/2})$ time with $O(S\log^{(c+1)} n)$ operations for each group ($O(n_3 \log^{(c+1)} n)$ operations for all linked lists). This factor of $\log^{(c+1)} n$ is introduced because of pointer jumping. We can remove this $\log^{(c+1)} n$ factor because we can pack $c(\log n)^{1/2}$ words in S_3 into one word and therefore the pointer jumping needs not to be done by every word in S_3. Thus the linked list contraction takes $O((\log n)^{1/2})$ time and $O(n_3)$ operations.

More complications of this process such as where to store dummy elements, how to move words to sorted position, etc., are explained in [13].

Let us estimate the complexity. Because each stage removes $c(\log n \log\log n)^{1/2}$ bits. there are $O((\log n/\log\log n)^{1/2})$ stages. Because each stage takes $O((\log n)^{1/2})$ time the time for our algorithm in this section is $O(\log n/(\log\log n)^{1/2})$. Each stage takes $O(n_3)$ operations and therefore for all stages it has $O(n_3 \log n/(\log\log n)^{1/2}) = O(n)$ operations.

Now for each linked list L, the words of S_3 on L are all having the same value (i.e. the j-th integer in all these words are the same. However, the i-th integer and the j-th integer may be different.). We can group every $(\log n/\log\log n)^{1/2}$ words on L together and do a transposition (put the j-th integer in all these words in one word). This takes $O(\log\log n)$ time and $O(n_3 \log\log n)$ operations (this should be simple and readers can work it out or see [13]). After that we sort the transposed words into linked lists in $O((\log n\log\log n)^{1/2})$ time and $O(n_3(\log\log n)^{1/2})$ operations using Lemma 2 (note that now each word contains $(\log n/\log\log n)^{1/2}$ integers of the same value which is in $\{0, 1, ..., 2^{c(\log n\log\log n)^{1/2}} - 1\}$).

Thus we have:

Theorem 1: n integers in $\{0, 1, ..., 2^{c(\log n\log\log n)^{1/2}}\}$ can be sorted into linked lists on the CREW PRAM with word length $\log n$ in $O(\log n/(\log\log n)^{1/2})$ time and $O(n)$ operations. □

4 Sorting Integers in $\{0, 1, ..., n^{1/2}\}$ and in $\{0, 1, ..., n-1\}$

To sort n integers in $\{0, 1, ..., n^{1/2}\}$ We apply Theorem 1 $(1/(2c))(\log n/\log\log n)^{1/2})$ times and reach

Theorem 2: n integers in $\{0, 1, ..., n^{1/2}\}$ can be sorted on the CREW PRAM with word length $\log n$ in $O((\log n)^{3/2}/\log\log n)$ time and $O(n(\log n/\log\log n)^{1/2})$ operations. □

The situation for sorting n integers in $\{0, 1, ..., n-1\}$ is different. For sorting the most significant $\log n/2$ bits we can apply Theorem 2. After that n integers are partitioned into $n^{1/2}$ sets and we have to sort every set concurrently and independently. Here on the average each set has $n^{1/2}$ integers. When we are sorting $n^{1/2}$ integers we cannot pack every $(\log n/\log\log n)^{1/2}$ integers to form words of $\log n$ bits in paragraph 2 of Section 3 (if the algorithm in Section 3 is well understood then one can see that n integers corresponds to $\log n$ bits for

sorting). To sort $n^{1/2}$ integers we can use only $\log n/2$ bits and therefore we can pack only $(1/2)(\log n/\log\log n)^{1/2}$ integers in $\{0,1,...,2^{c(\log n \log\log n)^{1/2}}\}$ into one word. However, because the number of bits is reduced by half the number of stages in the algorithm of Theorem 1 is also reduced by half. Thus sorting the next $\log n/4$ bits has half the time complexity but the same operation complexity as sorting the most significant $\log n/2$ bits. Again sorting the next $\log n/8$ bits takes the $1/4$ time complexity and the same operation complexity as sorting the most significant $\log n/2$ bits.

Thus if we iterate t times we spend $O((\log n)^{3/2}/\log\log n)$ time and $O(tn(\log n/\log\log n)^{1/2})$ operations and we have $\log n/2^t$ bits left to be sorted. By Lemma 2 the remaining $\log n/2^t$ bits can be sorted in $O((\log n)^{1/2}/2^t)$ time and $O(n(\log n)^{1/2}/2^t)$ operations. Now to pick the optimal t let

$$tn(\log n/\log\log n)^{1/2} = n(\log n)^{1/2}/2^t$$

and we obtain that $t = (\log\log\log n)/2$. Thus we have that

Theorem 3: n integers in $\{0,1,...,n-1\}$ can be sorted on the CREW PRAM with word length $\log n$ in $O((\log n)^{3/2}/\log\log n)$ time and $O(n(\log n/\log\log n)^{1/2}\log\log\log n)$ operations. □

5 Conclusions

We presented a CREW integer sorting algorithm which outperforms the operation complexity of previous best result. Many problems remains open such as: can we remove concurrent read from our algorithm? can time complexity be lowered to $O(\log n)$? can we sort integers with value larger than n? etc.. Note that Han proved before [11] that n integers in $\{0,1,...,m-1\}$ can be sorted on the EREW PRAM in $O((\log n)^2)$ time and $O(n(\log\log n)^2\log\log\log n)$ operations provided that $\log m \geq (\log n)^2$. This provides a partial solution to one of the open problems mentioned here. Our hunch is that removing the restriction of integers being bounded by n probably should be the next target to achieve. We hope our future research will resolve some of the open problems mentioned here.

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, T. Hagerup, S. Nilsson, R. Raman, Sorting in linear time? *Proc. 1995 Symposium on Theory of Computing*, 427-436(1995).
4. V.E. Benes, On rearrangeable three-stage connecting networks, *Bell Syst. Tech. J.*, Vol. 41, 1481-1492(1962).
5. V.E. Benes, Mathematical Theory of Connecting Networks and Telephone Traffic, New York: Academic, 1965.

6. S. Chen and John H. Reif. Using difficulty of prediction to decrease computation: fast sort, priority queue and convex hull on entropy bounded inputs. *34th Annual IEEE Conference on Foundations of Computer Science (FOCS '93) Proceedings*, November 1993, Palo Alto, CA, pp. 104-112.
7. S. Cook, C. Dwork, and R. Reischuk. Upper and Lower Time Bounds for Parallel Random Access Machines without Simultaneous Writes. *SIAM J. Comput.* Volume 15, Issue 1, pp. 87-97 (1986).
8. 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).
9. 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).
10. Y. Han, An optimal linked list prefix algorithm on a local memory computer, *Proc. 1989 Computer Science Conference (CSC'89)*, 278-286(Feb., 1989).
11. Y. Han. Improved fast integer sorting in linear space. *Information and Computation*, Vol. 170, No. 1, 81-94(Oct. 2001).
12. Y. Han. Deterministic sorting in $O(n \log\log n)$ time and linear space. *Journal of Algorithms*, 50, 96-105(2004).
13. Y. Han, X. Shen. Parallel integer sorting is more efficient than parallel comparison sorting on exclusive write PRAMs. *SIAM J. Comput.* 31, 6, 1852-1878(2002).
14. W.L. Hightower, J. Prins, and John H. Reif. Implementations of randomized sorting on large parallel machines. *4th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'92)*, San Diego, CA, pp. 158-167, July 1992.
15. J. JáJá, An Introduction to Parallel Algorithms. Addison-Wesley, 1992.
16. T. Leighton, Tight bounds on the complexity of parallel sorting, *IEEE Trans. Comput.* C-34, 344-354(1985).
17. John H. Reif. An $n^{1+\epsilon}$ processor, $O(\log n)$ time probabilistic sorting algorithm. *SIAM 2nd Conference on the Applications of Discrete Mathematics*, Cambridge, MA, June 1983, pp. 27-29.
18. S. Rajasekaran and John H. Reif. An optimal parallel algorithm for integer sorting. *26th Annual IEEE Symposium on Foundations of Computer Science*, Portland, OR, October 1985, pp. 496-503. Published as "Optimal and sublogarithmic time randomized parallel sorting algorithms." *SIAM Journal on Computing*, Vol. 18, No. 3, June 1989, pp. 594-607.
19. L.G. Valiant and John H. Reif, A Logarithmic Time Sort for Linear Size Networks. *15th Annual ACM Symposium on Theory of Computing, Boston, MA, April 1983, pp. 10-16. Published in Journal of the ACM(JACM)*, Vol. 34, No. 1, January 1987, pp. 60-76.
20. J. C. Wyllie, The complexity of parallel computation, *TR 79-387, Department of Computer Science, Cornell University*, Ithaca, NY, 1979.

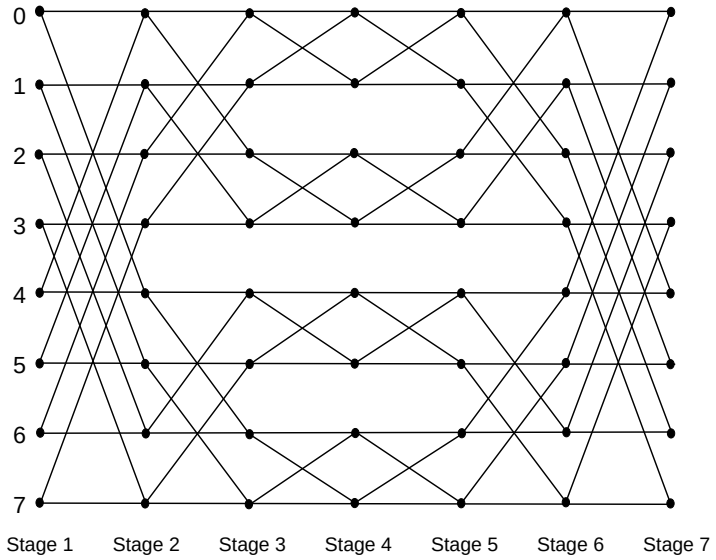

Fig. 1. A permutation network.

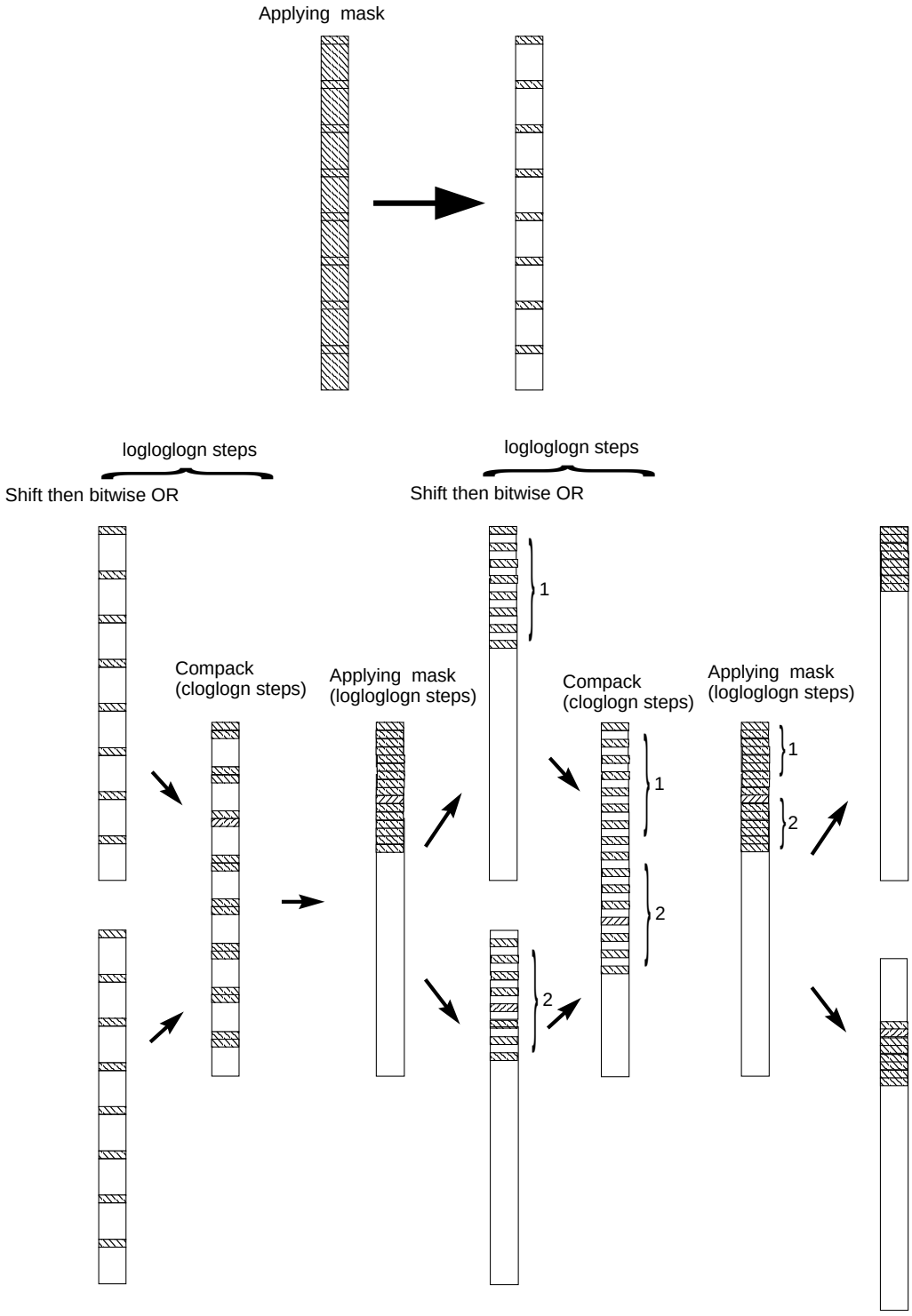

Fig. 2. Packing.