k-Integer-Merging on Shared Memory

Ahmed Y Khedr[1], Ibrahim M Alseadoon[2]
College of Computer Science and Engineering, University of Ha'il, Ha'il, KSA[1]
Systems and Computers Department, Faculty of Engineering, Al-Azhar University, Cairo, Egypt[1]
College of Computer Science and Engineering, University of Ha'il, Ha'il, KSA[2]

***Abstract*— The k integer-merging problem is to merge the k sorted arrays into a new sorted array that contains all elements of $A_i, \forall\, i$. We propose a new parallel algorithm based on exclusive read exclusive write shared memory. The algorithm runs in $O(\log n)$ time using $n/\log n$ processors. The algorithm performs linear work, $O(n)$, and has optimal cost. Furthermore, the total work done by the algorithm is less than the best-known previous parallel algorithms for k merging problem.**

***Keywords*—Merging; parallel algorithm; shared memory; optimality; linear work**

I. Introduction

The problem of merging has many applications in computer science and used as a subroutine for solving many problems such as sorting [1], database management systems [2], information retrieval [3], memory management, scheduling [1], and reconstruction of the tree [4][5]. Most of these applications based their solutions on the merging problem. For example, the optimal algorithm for sorting an array *A* of size *n* can be done as follows. (1) Partition the array *A* into two subarrays of equal size, A_1 and A_2. (2) Sort recursively for A_1. (3) Sort reclusively for A_2. (4) Merge A_1 and A_2.

The merging problem is defined as follows [10]. Given two sorted arrays $A = (a_0, a_1,..., a_{n-1})$ and $B = (b_0, b_1,..., b_{m-1})$. The merging of two sorted arrays is a new sorted array $C = (c_0, c_1,.. ., c_{n+m-1})$ such that:

1) $c_i \in A$ or $c_i \in B$, $\forall\, 0 \le i \le n + m - 1$; and

2) a_i and b_j appear exactly once in *C*, $\forall\, 0 \le i < n$ and $0 \le j < m$.

On the other side, some applications of computer science such as external sorting and information retrieval systems require to merge *k* sorted arrays of different lengths. In such a case, the problem is known as a *k* merging problem. For example, the external sorting (sorting a file of large data) problem can be done by the following steps [1]: (1) Dividing the file into small blocks to fit into main memory. (2) Applying the fast sorting algorithm on each block. (3) Merging the sorted blocks into sorted bigger blocks, until the file is sorted.

The merging problem of *k* sorted arrays is defined as follows.

Given k sorted arrays of lengths $n_i's$, $A_i = (a_{i\,0}, a_{i\,1}, \cdots, a_{i\,j}, \cdots, a_{i\,n_i-1})$, such that $\sum_{i=0}^{k-1} n_i = n$, $0 \le i < k$, $0 \le j < n_i$, and $2 \le k \le n$. The merging of k sorted arrays is a new sorted array $C = (c_0, c_1, \ldots, c_{n-1})$ such that:

1) c_j belongs to one of A_i, $\forall\, 0 \le j < n$; and

2) $a_{i\,j}$ appear exactly once in *C*, $\forall\, 0 \le i < k$ and $\forall\, 0 \le j < n_i$.

In sequential computation, the problem of merging two arrays of sorted elements is solved in linear time, $O(n)$, where $n \ge m$ [1][7], while the problem of *k* merging required $\Omega(n \log k)$, where $2 \le k \le n$ [17].

In parallel computation, different algorithms solve the problem of computation based on different strategies and parallel computational models. The two main types of parallel models are shared memory and interconnection networks. Our paper focuses only on the type of shared memory that is the Parallel Random Access Machine, PRAM. PRAM consists of *p* identical processors that operate p synchronously and communicate through large shared memory. There are three main models for PRAM based on memory access conflicts in shared memory. (1) Allowing read or write operations to memory location that is called Exclusive Read Exclusive Write (EREW). (2) Allowing only read operations to a memory location that is called Concurrent Read Exclusive Write (CREW). (3) Allowing read or write operations to a memory location that is called Concurrent Read Concurrent Write (CRCW).

In case of merging two sorted arrays [9] [10] [11][12] [13] [14] [15], the optimal parallel algorithm uses $p \le n/\log n$ processors to run the algorithm in $O(n/p)$ time. The algorithm is based on EREW PRAM [12]. In case of CREW, Kruskal [13] shows that the merging problem can be solved in $O(\log \log n)$ time using $O(n)$ work.

In case the elements of the two arrays are taken from integer domain $[1,m]$, then the problem of merging is called integer merging [6][16][17][18][19]. The problem of integer merging needs further investigation. In case of EREW PRAM, Hagerup and Kutyloushi [19] presented $O(\log \log n + \log \min\{n, m\})$ algorithm with total space $O(n)$, where $m =$

Deanship of Scientific Research, University of Ha'il

$O(n)$. The total number of operations done by the algorithm is $O(n)$. Additionally, Bahig [16][17] reduced running time to constant, by considering some properties for the elements of the input. These properties are: (1) each element has a constant number of repetitions and (2) the difference between two successive elements is bounded by a constant. In case of CREW PRAM, Berkman and Vishkin in [6] proposed an algorithm that uses $n/\log\log\log m$ processors and has running time $O(log\ log\ log\ m)$, when $m = n^k$. Also; they have proposed $O(\alpha(n))$ algorithm by using $n/\alpha(n)$ processors, where $\alpha(n)$ is the inverse of Ackermann's function and m=n. Furthermore the author in [18] proposed a constant-time deterministic algorithm, $O(1)$, for merging on CREW. The proposed algorithm is optimal in case of the values of input elements are less than or equal to the size of the inputs and the number of processors is equal to size of the inputs.

In case of k merging problem, PRAM is the main used algorithm. In [19], the algorithm is based on repeated pairwise merging of k sorted arrays. The algorithm is not working optimally and it is running in $O(\log n \times \log k)$ time. In [20], the author proposed an algorithm based on CREW PRAM. The algorithm has $O(log\ n)$ parallel time using $(n\ log\ k)/log\ n$ processors with total work $O(n\ log\ k)$. The algorithm based its solution on pipelining strategy and optimal work. In [8], the authors proposed two optimal parallel algorithms on PRAM with work $O(n\ log\ k)$. Previous two algorithms are based on sampling scheme. The first algorithm runs in $\Omega(log\ n)$ under EREW, while the second algorithm runs in $\Omega(log\ log\ n + log\ k)$ under CREW. Recently, the authors in [24] proposed a lazy-merge algorithm, have running time equal to O(k *log* (n/k) + merge(n/p)), where k and merge(n/p) are the number of segments and the time needed to merge n/p elements respectively by the used in-place merging algorithm. Also, the authors in [21] presented two parallel algorithms for k integer merging, when no repetition occurs in the elements. The running time for both algorithms are $O(log\ n)$ and $O(1)$ under EREW and CREW PRAM respectively.

The paper studies the k integer merging problem on PRAM and shows that the k integer merging problem can be solved in total work $O(n)$, even though the elements number of repetitions is extreme. The proposed algorithm uses $n/log\ n$ processors of type EREW PRAM to run the algorithm in time $O(log\ n)$.

The paper is organized as follows: an introduction and five sections. In Section 2, we give foundations and subroutines that is needed for proposed algorithm. Proposed algorithm for k integer merging is explained in Section 3. In Section 4, we calculate the complexity analysis of the proposed algorithm. In Section 5, we show how the proposed algorithm works by tracing the algorithm on an example. Finally, in Section 6, we show the conclusion of our work.

II. Preliminaries

In this section, we give the fundamental definitions and subroutines related to k integer merging problem.

Definition 1 [10][22]: Given a problem Q of size n. The cost of the parallel algorithm for Q is equal to the product of the number of processor used and the running time for the parallel algorithm.

Definition 2 [10][22]: Given a problem Q of size n. The cost of parallel algorithm for Q is optimal if it matches with the time complexity of the best-known sequential algorithm for Q.

Definition 3 [8][10][22]: The work of a parallel algorithm is the total number of operations that the processors perform.

Definition 4 [9][10]: The prefix sums of the array $A = (a_0, a_1, \ldots, a_{n-1})$ is an array $S = (s_0, s_1, \ldots, s_{n-1})$, where $s_i = a_0 \oplus a_1 \oplus \ldots \oplus a_i$, and $\oplus$ is a binary operator, $\forall\ 0 \le i < n$.

Proposition 1 [23]: An EREW PRAM algorithm for computing the prefix sums of an array of n elements runs in $O(n/p)$ time using p, $p \le n/\log n$.

The technique used to solve the prefix sum is called binary tree strategy and it can be used to solve many related problems [9][23].

Proposition 2 [1][9][10]: Given an array A of n integers. The integer sorting algorithm runs in $O(n)$ sequential time and $O(\log n)$ using $n/\log n$ time under EREW PRAM.

III. New Parallel Algorithm

In this section, we show that the k integer merging problem on exclusive read exclusive write shared memory model can be solved in total work $O(n)$ instead of $O(n \log k)$, which is the total work for the best-known algorithm for k merging on the same shared memory model. Without loss of generality, assume that the elements of the k sorted arrays, $a_{i\ j}$, are taken from the integer domain [0,n-1], $\forall\ 0 \le i < k, 0 \le j < n_i$, $2 \le k \le n$, and $n = \sum_{i=0}^{k-1} n_i$. The elements of the k sorted arrays are uniformly distributed over the integer domain. We also assume that the number of processors used to design the parallel algorithm is $n/log\ n$.

The main idea behind the proposed algorithm is how to partition the k sorted arrays into $n/log\ n$ independent lists. Then, the proposed algorithm assigns each list to a processor to merge it sequentially. The algorithm consists of the following stages.

A. Stage 1: Partitioning

The partitioning stage is the first stage to merge k sorted arrays of integer elements. The goal of this stage is to divide the elements of the k sorted arrays into $n/\log n$ lists. The lists have the following properties.

P1: The lists are independent which means that the elements of the list number i are different from the elements of the list number $j, \forall\, i \neq j$.
P2: The lists are relatively ordered which means that the elements of the list number i is less than the elements of the list number $j, \forall\, i < j$.
P3: A small integer range called bounded range, BR., bound the difference between the elements in a list.

To verify the three properties of the $n/\log n$ lists, we define the value of *BR* by the following equation:

$$BR = \begin{cases} \log n & \text{if } n/\log n \text{ is perfect integer} \\ \log n + 1 & \text{otherwise} \end{cases}$$

To construct the independent lists, we will use two phases to partition the *k* sorted arrays. These two phases are called *local* and *global* partitioning.

B. Local Partitioning Phase

The main objective of the local partitioning phase is to partition each sorted array A_i into many subarrays based on the values of the elements of A_i. The elements of the *i*th subarray belong to the range $[(i-1)BR, i\,BR-1], \forall\, 1 \leq i \leq n/\log n$.

The input of local phase is *k* sorted arrays $A_0, A_1, \ldots, A_{k-1}$ of lengths $n_0, n_1, \ldots, n_{k-1}$, respectively. By the end of the local partitioning phase, we construct a list, AL_i, of l_i elements for each sorted array A_i, $\forall\, 0 \leq i < k$. The list contains the boundary indices for each partition in the sorted array A_i. Each element in the list AL_i, $AL_i[j]$, consists of four fields, *aNo*, *pNo*, *start* and *end*. The component *aNo* represents the array number, while the component *pNo* represents the partition number. The fields *start* and *end* represent start and end indices for the partition number *pNo* in the sorted array A_i, respectively.

To construct the elements of AL_i, we have two cases. The first case is when the size of each sorted array is approximately equal to *BR*. The second case is when the sizes of the sorted arrays are different.

In the case, the size of each sorted array A_i is approximately equal to *BR*, we do Subroutine 1 to construct the local partition. In the subroutine, we can compute the component of each element for the list AL_i by the following steps.

Initially, the number of elements in the list AL_i is equal to 0, and the first element in the array A_i, a_{i0}, determines the first partition belong to the list AL_i, see lines 1-2 in Subroutine 1.

The first three components of $AL_i[0]$ are as follows:

$A\Lambda_\iota[\lambda_\iota]\cdot\, \alpha No = \iota$

$A\Lambda_\iota[\lambda_\iota]\cdot\, \pi No = \alpha_{\iota 0}\ \Delta\iota\varpi\ BP$

$A\Lambda_\iota[\lambda_\iota]\cdot\, \sigma\tau\alpha\rho\tau = 0$

The partition number, *pNo*, is determined by using the Div operator to return the quotient of division. The fourth component will be determined later when the algorithm determines the start of a new partition. Then, the algorithm scans the elements of the array A_i from the second to the last elements to determine the end of the current partition and the start of a new partition (see line 3 in Subroutine 1). The end and the start of the current and new partitions, respectively, can be determined by testing if the quotients for the two successive elements, $a_{i\,j-1}$, and $a_{i\,j}$, on BR are different. When, the result of comparison is different, then, the partition number, *pNo*, is equal to $a_{i\,j}$ Div *BR*. In such case, the element $a_{i\,j}$ represents the first element of a new partition and the index *j* represents the start index of current partition. On the other side, the element $a_{i\,j-1}$ represents the last element of the current partition and the index *j*-1 represents the last index of the current partition.

In case, the sizes of the *k* sorted arrays are different; we can compute $AL_{i,}\forall i$, using two steps. The first step include that, we take each array of size greater than or equal to *BR* and do the following:

1) Determine the number of processors required for the array A_i which is equal to $np_i = \lfloor n_i/BR \rfloor$.
2) Each processor, p_j, will do the same process as in Subroutine 1 on the *i*th partition of AL_{ij}, $\forall\, 0 \leq j < np_i$.
3) Combine all the sublists, AL_{ij}, to the list AL_i by using the binary tree paradigm.

After finishing from all arrays of sizes greater than or equal to *BR*, we execute the second step. The second step includes that, we assign one processor to each of the remainder arrays and do the same process as in Subroutine 1.

Subroutine 1:
Each processor do the following:
1. l_i=0 // number of elements in AL_i
2. $AL_i[l_i]$=(i,a_{i0} Div *BR*,0,)
3. for j=1 to n_i -1 do
4. if $a_{i\,j}$ Div *BR* ≠ $a_{i\,j\text{-}1}$ Div *BR* then
5. $AL_i[l_i]$=(, , ,j-1)
6. l_i= l_i+1
7. $AL_i[l_i]$=(i,$a_{i\,j}$ Div *BR*,j,)
8. end if
9. end for
10. $AL_i[l_i]$=(, , ,n_i-1)

C. Global Partitioning Phase

The main objective of the global partitioning phase is to partition the *k* sorted arrays into $n/\log n$ lists. Each list satisfies the three previous properties (mentioned in Stage1).

The input of global phase is a collection of *k* lists $AL_0, AL_1, \ldots, AL_{k-1}$, of lengths $l_0, l_1, \ldots, l_{k-1}$, respectively. By the end of this phase, we have an array, *AP*, of $n/\log n$ elements. The element *AP*[*i*] consists of three fields. The first two fields, *start* and *end*, represent the start and the end indices for the elements of the *k* sorted arrays that belong to the partition number *i*. The third field, *no*, represents the number of elements in the partition *i*, for all *k* sorted arrays. We can construct the array *AP* as follows:

1) Apply the parallel integer sort algorithm [24] on the elements of *k* lists, $AL = \cup_{i=0}^{k-1} AL_i$, according to the second

component of AL_i, *pNo*, using $n/\log n$ processors. If there are two elements in *AL* having the same value of the second component, then the elements are ordered according to the first component. The output of the first step is an array *AL* of length $l = \sum_{i=0}^{k-1} l_i \leq n$.

2) Divide *AL* into $n/\log n$ partitions of approximately equal size, $(l * \log n)/n$.

3) Initially, compute the start and the end of the first and the last partitions, respectively, as follows:

$$AP[AL[0] \cdot pNo] \cdot start = 0$$

$$AP[AL[l-1] \cdot pNo] \cdot end = l - 1$$

4) Determine the start and the end of each list that satisfies our proposed three properties, by applying Subroutine 2 on each partition.

5) Each processor, p_i, determines the third component of the partition, $AP[i]$, by scanning the array *AP* from $AL[i] \cdot start$ to $AL[i] \cdot end$ and calculates the total number of elements using the following formula:

$$AP[i] \cdot no = \sum_{j=AP[i] \cdot start}^{AP[i] \cdot end} AL[j] \cdot end \; - AL[j] \cdot start + 1$$

Subroutine 2

Each processor p_i do the following test on its partition as follows: $0 \leq i < p$ and $i\lceil l/p \rceil \leq j < (i+1)\lceil l/p \rceil$, except i=0 and j=0.

1. if $AL[j] \cdot pNo \neq AL[j-1] \cdot pNo$ then
2. $AP[AL[j] \cdot pNo] \cdot start = j$
3. $AP[AL[j-1] \cdot pNo] \cdot end = j - 1$

Note that in case $i=p-1$, the value of j is less than n.

D. Stage 2: Merging

The main objective of the merging stage is to merge elements of each partition. In other words, the goal is to merge the sorted subarrays that belong to the *i*th partition, $AP[i]$.

To merge sorted subarrays that belong to the *i*th partition, we have two cases based on the number of elements in each partition. In the first case, the size of each partition is approximately equal to BR, while in the second case the size of each partition is different.

In the first case, we do the process of merging by using Subroutine 3, which uses the idea of counting sorting algorithm [25]. To verify our goal, we use an array CA_i of length *BR* to merge the subarrays that belong to the partition number *i*, $\forall\, 0 \leq i < n/\log n$. Each element in this array consists of two fields. The first component, *val*, represents the value of the element, while the second component, *count*, represents the number of repetitions of the element *val*. The first step of Subroutine 3 is to initialize the two fields of the array CA_i with $i \log n + j$ and 0, respectively as in lines 1-3 in Subroutine 3. In the second step we compute the number of repetitions for each element by traversing the elements of the partition $AP[i]$ in lines 4-7 in Subroutine 3. In the third step, we reallocate the elements of the auxiliary array CA_i to the array C_i.

In case that the size of each partition is different, we can construct CA_i by the same method that is described in local partitioning step.

Subroutine 3

Processor p_i do the following

1. for j=0 to *BR*-1 do
2. $CA_i[j] \cdot val = i \log n + j$
3. $CA_i[j] \cdot count = 0$
4. for $j = AP[i] \cdot start$ to $AP[i] \cdot end$ do
5. for $x = AL[j] \cdot start$ to $AL[j] \cdot end$ do
6. $y = AL[j] \cdot aNo$
7. $CA_i\left[A_y[x] \bmod \log n\right] \cdot count = CA_i\left[A_y[x] \bmod \log n\right] \cdot count + 1$
8. x=0
9. for j=0 to *BR*-1 do
10. while $CA_i[j] \cdot count \geq 1$ do
11. $C_i[x] = CA_i[j] \cdot val$
12. $CA_i[j] \cdot count = CA_i[j] \cdot count - 1$
13. $x = x + 1$

IV. Complexity Analysis

In this section, we analyze the proposed parallel algorithm for k integer merging problem according to the following criteria: running time, total number of work, optimality, and storage.

To compute the running time of the parallel proposed algorithm, the algorithm consists of three main stages: local partitioning, global partitioning, and merging.

The running time for the local partitioning stage can be computed as follows. In case that the size of each A_i is approximately equal to BR, each processor p_i will execute a sequential loop on an array of length $O(BR)$ approximately.

Therefore, the running time of this step is $O(BR)$=O($log\ n$). In case of the size of A_i is different, the running time can be computed as follows. Determining the number of processors that is required for A_i equal to constant time. The running time for step 2, execution of Subroutine 1, and step 3, combine all the sublists, are $O(BR)$ and $O(\log np_i)$, respectively. The overall time for the local partitioning phase is $O(BR + np_i) = O(log\ n)$.

The running time for the global partitioning stage can be computed as follows. The running time for applying the parallel integer sort algorithm on *AL* is bounded by $O(\log n)$, because the maximum length of the list *AL* is *n*. The running time for the substep 2.1 is constant. The running time for the substep 2.2 is $O(\log n)$, because $\frac{l * \log n}{n} \leq \log n$. The running time for the substep 2.3 is $O(\log(n/\log n))$. Therefore, the overall running time for global partitioning is $O(\log n)$.

The running time for the merging phase can be computed as follows. In case of the size of each $AP[i]$ is approximately

equal to BR, the running time for Step 3.1 and 3.3 in Subroutine 3, are $O(BR)$. The running time for Step 3.2 in Subroutine 3 depends on the size of the partition, which is equal to $O(BR)$. So, the overall time for Subroutine 3 is $O(BR)$. In case the size of each partition is different, then, the running time can be computed by a similar way which is equal to $O(\log n)$. The overall running time of the algorithm is $O(\log n)$.

It can be seen from previous calculation that the total number of work done by each processor p_i is $O(\log n), \forall\, 0 \leq i < n/\log n$. Hence, the algorithm has a total work of $O(n)$.

Therefore, the proposed algorithm has optimal work and cost. Also, the storage required by the proposed algorithm is $O(n)$.

Finally, it is clear that no step in the algorithm requires concurrent read or write. So, proposed algorithm based its work on exclusive read exclusive write shared memory.

V. EXAMPLE

Assume that we have six sorted arrays of total lengths equal to 32 as in Fig. 1.

It is clear that k=6, n=32, n_0=5, n_1=11, n_2=6, n_3=1, n_4=7, and n_5=2. Therefore, the number of processors required is $p = \lfloor n/\log_2 n \rfloor = 6$ and BR=5+1=6.

Now, we apply the first stage (local partitioning) on the six sorted arrays as follows. For the sorted array A_0, the details of constructing the list AL_0 are as follows. Initially, l_0=0 and $AL_0[0]$=(0,0,0,) because i=0 and 2 Div 6 =0. For j=1, no updating for AL_0 because 5 Div 6 =2 Div 6. For the next iteration, j=2, AL_0 will be updated as $AL_0[0]$=(0,0,0,1), l_0=1 and $AL_0[1]$=(0,2,2,) because 13 Div 6 ≠5 Div 6. For next iteration, j=3, no updating for AL_0 because 13 Div 6 =13 Div 6. For last value of j=3, the updating values of AL_0 are as follows. $AL_0[1]$=(0,2,2,3), l_0=2 and $AL_0[2]$=(0,3,4,) because 23 Div 6 ≠13 Div 6. Finally, the last component of the final element in AL_0 become $AL_0[2]$=(0,3,4,4). Hence, the elements of AL_0 are (0,0,0,1), (0,2,2,3), and (0,3,4,4). The results of applying the first stage on all sorted input arrays are as in Fig. 2.

Next, the algorithm starts to execute the global partitioning stage by sorting the elements of all lists, AL_0, AL_1, AL_2, AL_3, AL_4, and AL_5 to obtain a sorted list AL of l=17 elements as in Fig. 3.

Initially, the processor p_0 determines the start and the end of the first and last partitions, respectively, as follows:

$$AP[0] \cdot start = 0$$

$$AP[16] \cdot end = 16$$

After that, each processor assigned to three elements, except the last processor has two elements, to determine the first two components, *start* and *end* in the array *AP*. For more details, the first three elements, (0,0,0,1), (1,0,0,1) and (4,0,0,1), are assigned to the processor p_0. The second three elements, (1,1,2,4), (2,1,0,1), and (4,1,2,3), are assigned to the processor p_1, while the last two elements, (5,4,1,1) and (1,5,10,10), are assigned to the processor p_5.

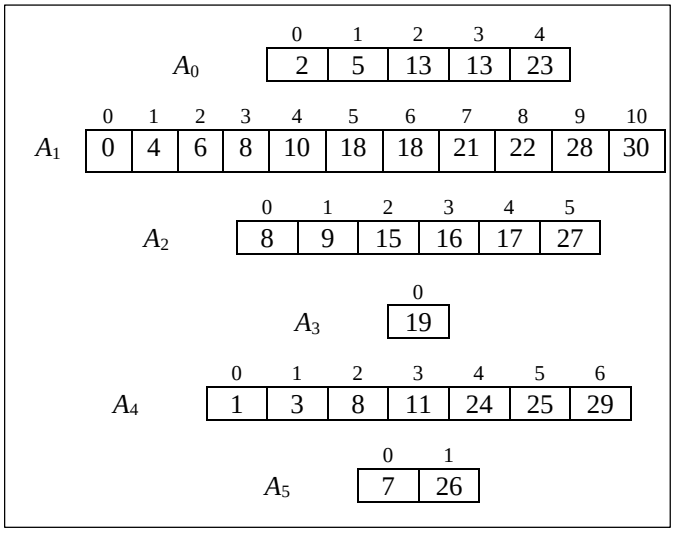

Fig. 1. Input Data.

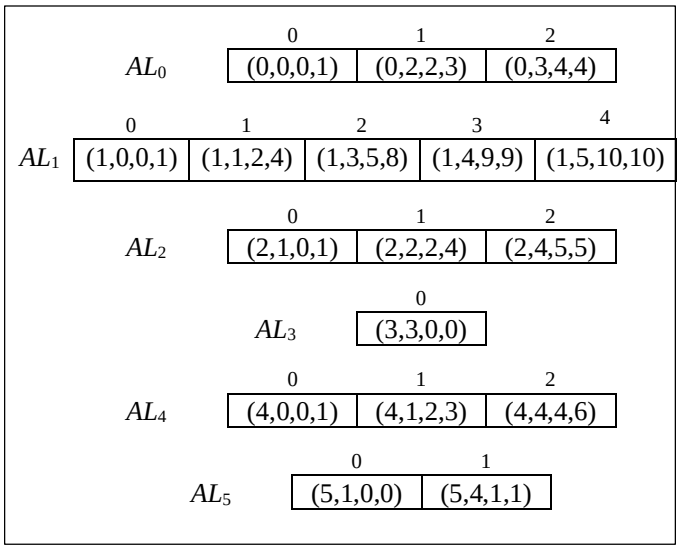

Fig. 2. Execution of Local Partition.

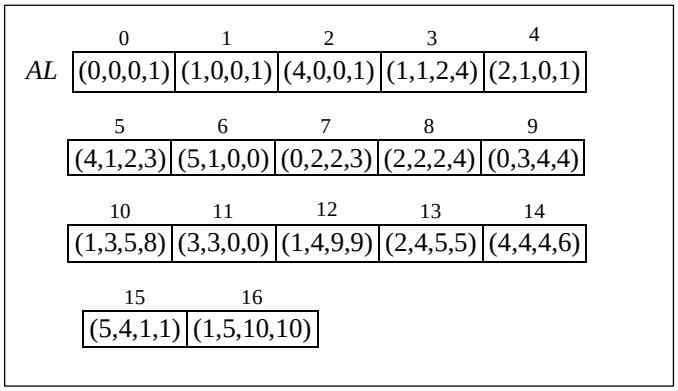

Fig. 3. First Step of Global Partition.

The processor p_0 does not determine any components because $AL[1] \cdot pNo \neq AL[0] \cdot pNo$ and $AL[2] \cdot pNo \neq AL[1] \cdot pNo$ are false. The processor p_1 determines the start of the second partition and the end of the first partition because

$AL[3] \cdot pNo \neq AL[2] \cdot pNo$ is true. So, $AL[1] \cdot start = 3$ and $AL[0] \cdot end = 2$. On the other side, the two other elements do not lead to new partition because $AL[4] \cdot pNo \neq AL[3] \cdot pNo$ and $AL[5] \cdot pNo \neq AL[4] \cdot pNo$ are false. The last processor p_5 determines the start of the fifth partition and the end of the fourth partition because $AL[16] \cdot pNo \neq AL[15] \cdot pNo$ is true. So, $AL[5] \cdot start = 16$ and $AL[4] \cdot end = 15$. The array AP is shown in Fig. 4(*a*).

By applying Step 5 of the global partition phase, each processor p_i computes the number of elements in each partition. For the processor p_0, $AP[0] \cdot pNo = 2 + 2 + 2 = 6$, while p_1 computes $AP[1] \cdot pNo = 3 + 2 + 2 = 7$. The array *AP* becomes as in Fig. 4(*b*).

In the last stage, each processor, p_i, merges the different subarrays of the *i*th partition. The results of implementing the last stage consist of two steps. In the first step, each processor computes the repetition of each element as shown in Fig. 5. The results of the second step are shown in Fig. 6. The output array is *C*=(C_0, C_1, C_2, C_3, C_4, C_5)= (0, 1, 2, 3, 4, 5, 6, 7, 8, 8, 8, 9, 10, 11, 13, 13, 15,16, 17, 18, 18, 19, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30).

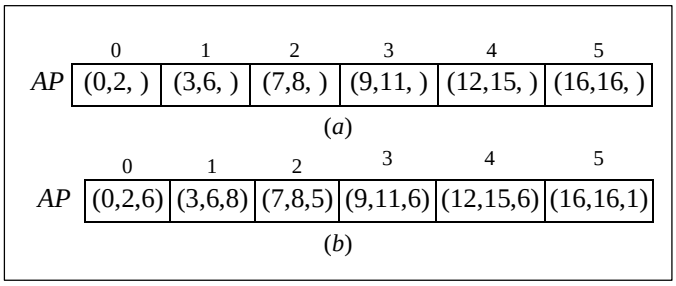

Fig. 4. Result of Global Partition.

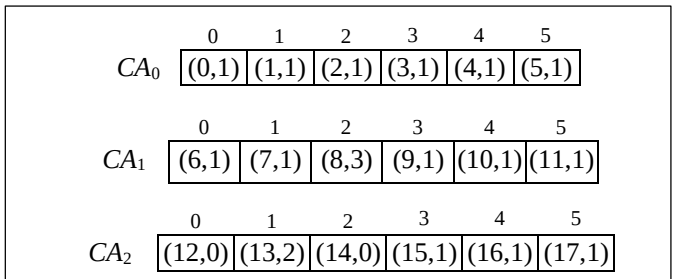

Fig. 5. First Step of Merging Phase.

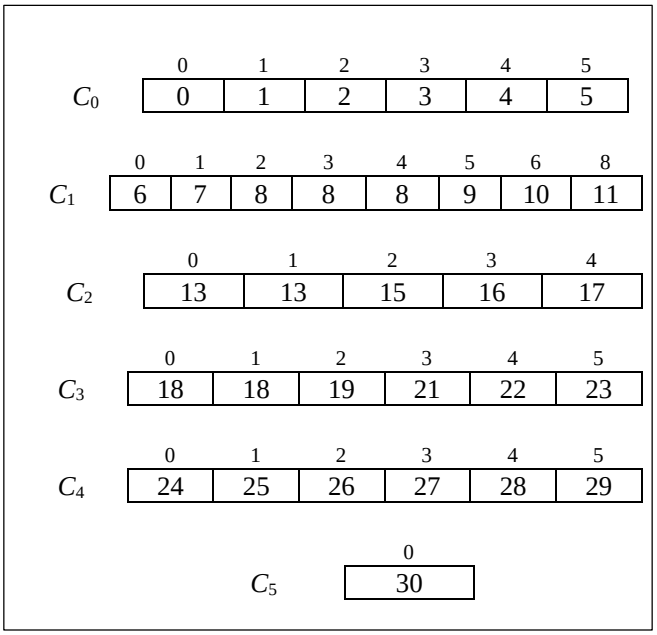

Fig. 6. Results of the Second Step of Merging Phase (Output).

VI. Conclusion

The paper addresses the problem of merging when the number of input sorted arrays is k, $2 \leq k \leq n$. The output of the merging is a new sorted array that contains all elements of the input. Our main contribution is solving the k integer merging problem under exclusive read exclusive write shared memory. The proposed algorithm runs in $O(\log n)$ time using $n/\log n$ processors. Additionally, the total work done by the proposed algorithm is $O(n)$, which is less than the best-known k merging parallel algorithm that perform $\Theta(n \log k)$ work.

References

[1] D. Knuth. The art of computer programming: sorting and searching. Addison–Wesley, Reading, 1973.

[2] P. Valduriez, G. Gardarin. “Join and semijoin algorithms for multiprocessors database machines”. ACM Transaction Database System Vol. 9, pp. 133–161, 1994.

[3] T. Merrett. Relational information systems. Reston Publishing Co., Reston, 1984.

[4] J. Bang-Jensen, et al. “Recognizing and representing proper interval graphs in parallel using merging and sorting”. Discrete Applied Mathematics, Vol. 155, No. 4, pp. 442–456, 2017.

[5] S. Olariu, et al. “Reconstructing binary trees in doubly logarithmic CREW time”. Journal of Parallel and Distributed Computing, Vol. 27, 1995, pp. 100-105.

[6] O. Berkman, U. Vishkin. “On parallel integer merging”. Information and Computation, Vol. 106, pp. 266–285, 1993.

[7] Th. Cormen, et al. Introduction to algorithms. MIT, Cambridge, 1990.

[8] T. Hayashi, et al. “Work-time optimal k-merge algorithms on the PRAM”. IEEE Transaction on Parallel and Distributed Systems, Vol. 9, No. 3, pp. 275-282, 1998.

[9] S. Akl. Parallel sorting algorithms. Academic Press, Orlando, 1985.

[10] S. Akl. Parallel computation: models and methods. Prentice Hall, Upper Saddle River, 1997.

[11] A. Borodin, and J. Hopcroft . “Routing, merging, and sorting on parallel models of computation”. Journal of Computer System Science, Vol. 30, pp. 130–145, 1995.

[12] T. Hagerup, and C. Rub. “Optimal merging and sorting on the EREW PRAM”. Information Processing Letters, Vol. 33, pp. 181–185, 1989.

[13] C. Kruskal. “Searching, merging, and sorting in parallel computation”. IEEE Transaction on Computers, Vol. 32, No. 10, pp. 942–946, 1993.

[14] S. Nagaraja, et al. “A parallel merging algorithm and its implementation with JAVA threads”. In: The Mid-Atlantic student workshop on programming languages and systems, IBM Watson Research Centre, 27 April 2001, 2001.

[15] A. Salah, et al. “Lazy-Merge: A Novel Implementation for Indexed Parallel K-Way In-Place Merging”. IEEE Transaction on Parallel and Distributed Systems, Vol. 27, No. 7, pp. 2049-2061, 2016.

[16] H. Bahig. “Parallel merging with restrictions”. The journal of Supercomputing, Vol. 43, No. 1, pp. 99-104, 2018.

[17] H. Bahig. “Integer Merging on PRAM”. Computing, Vol. 91, No. 4, pp. 365-378, 2011.

[18] H. Bahig. “A new constant-time parallel algorithm for merging”. The journal of Supercomputing, Vol. 72, No. 2, 968–983, 2019.

[19] T. Hagerup, M. Kutylowski. “Fast integer merging on the EREW PRAM”. Algorithmica, Vol. 17, pp. 55–66, 1997.

[20] Z. Wen. “Multi-Way Merging in Parallel”. IEEE Transaction on Parallel and Distributed Systems, Vol. 7, No. 1, pp. 11–17, 1996.

[21] B, Hazem and K, Ahmed. “Parallelizing K-Way Merging”. International Journal of Computer Science and Information Security, Vol. 14, No. 4, pp. 497-503, 2016.

[22] R. Karp, V. Ramachandran. “Parallel algorithms for shared-memory machines”. In: Van Leeuven J (ed) Handbook of theoretical computer science, Vol A: Algorithms and complexity. Elsevier, Amsterdam, 869–941, 1990.

[23] G. Blelloch. “Prefix sums and their applications”. TR CMU-CS-9-190, Carnegie Mellon University, 1990.

[24] S. Albers, T. Hagerup. “Improved Parallel Integer Sorting without Concurrent Writing”. Information and Computation, Vol. 136, No. 1, pp. 25-51, 1997.

[25] H. Bahig. “Complexity analysis and performance of double hashing sort algorithm”. Journal of the Egyptian Mathematical Society, 27, Article number 3, 2019.