Lazy-Merge: A Novel Implementation for Indexed Parallel K-Way In-Place Merging

Ahmad Salah, Kenli Li, *Member, IEEE*, and Keqin Li, *Fellow, IEEE*

Abstract—Merging sorted segments is a core topic of fundamental computer science that has many different applications, such as n-body simulation. In this research, we propose Lazy-Merge, a novel implementation of sequential in-place k-way merging algorithms, that can be utilized in their parallel counterparts. The implementation divides the k-way merging problem into t ordered and independent smaller k-way merging tasks (partitions), but each merging task includes a set of scattered ranges to be merged by an existing merging algorithm. The final merged list includes ranges with ordered elements, but the ranges themselves are not ordered. Lazy-Merge utilizes a novel usage of indexes to access the entire set of merged elements in order. Its merging time complexity is $O(k \log(n/k) + merge(n/p))$, where k, n, and p are the number of segments, the list size and the number of processors (partitions), respectively. Here, $merge(n/p)$ represents the time needed to merge n/p elements by the used in-place merging algorithm. The time complexity of accessing an element in the merged list is $O(\log k)$, that time can be constant if k processors are used. The results of the proposed work are compared with those of bitonic merge and the best time-space optimal algorithms on number of moves and execution time. In comparison with the existing algorithms, significant speedup and reasonable reduction factor for number of moves have been achieved.

Index Terms—In-place merging, *k*-way merging, lazy-merge, parallel algorithm, rosetta index

1 Introduction

MERGING sorted segments is a significant topic in computer science. It is an imperative step in the well-known merge-sort sorting algorithm [1], which plays a vital role in abundant applications. Merging is also a basic step in other applications, such as discovering proper interval graphs [2], optimizing n-body simulations [3], managing data cluster environments [4] and addressing Boolean queries in information retrieval systems [5]. Throughout the remainder of this paper, we refer to this problem as the merging problem, and every mentioned segment in the following is a sorted one.

Generally, the merging algorithms have a set of three properties, as detailed in [1]. The first property is stability; an algorithm is stable if it maintains the relative order of equal elements from the input segment through the sorted output segment. The second property is that it is in-place; the algorithm is in-place if it completes the merging using only a constant or trivial amount of space plus the space of the input segments; otherwise, it is out-of-place. The third property is simplicity vs. complexity; the algorithm is simple if it is easy to understand and implement due to its simple structure; otherwise, it is complex.

k-way merge is the algorithm that considers an input of k sorted segments. Its output is a single sorted list of all the elements. Regarding in-place merging, most of the literature addresses the sequential 2-way merging problem. However, less research addresses the k-way in-place merging problem via sequential [6] and parallel implementations [7], [8].

Intuitively, a recursive linear time and sequential 2-way in-place merging algorithm [9] can accomplish k-way merging in $O(\log(k) \times n)$ time and constant extra space, where $\log(k)$ presents the number of phases to traverse a tree from the leaves to the root. The parallelism level of this approach is limited as the algorithm moves towards the highest level, the root; it uses a single thread to merge the final two segments at the last level, i.e., the parallelism level is halved each time the algorithm traverses a level higher towards the root. These recursive calls form a bottom-up tree traversal, as we start with k segments, the leaves, and end with a single merged segment, the root.

Throughout the remainder of this paper, we refer to this recursive 2-way merging process as the *binary merging tree* because the k segments are merged by incremental 2-way merging tasks, as illustrated in Fig. 1 upper part.

A better approach is to use a binary merging tree and a parallel 2-way in-place merging algorithm [7], [8]. Thus, the parallelism level is not affected during the upward traversal. For example, merging k segments using the binary merging treeand the algorithm proposed in [7] can be completed in $O(\log(k) \times (n/p + \log(n)))$, for $p \leq \log(n)$, where k, n, and p are the numbers of segments, elements and processors, respectively. A binary merging tree via a merging network [10] is a similar attempt, but it is limited to segment

- *A. Salah is with the College of Information Science and Engineering, Hunan University, Changsha, Hunan, China, the National Supercomputing Center in Changsha, Hunan, China, and the Computer Science Department, College of Computers and Informatics, Zagazig University, Zagazig, Egypt. E-mail: ahmad@hnu.edu.cn.*
- *K. Li is with the College of Information Science and Engineering, Hunan University, Changsha, Hunan, China, and the National Supercomputing Center in Changsha, Hunan, China. E-mail: lkl@hnu.edu.cn.*
- *K. Li is with the College of Information Science and Engineering, Hunan University, Changsha, Hunan, China, the National Supercomputing Center in Changsha, Hunan, China, and the Department of Computer Science, State University of New York, New Paltz, NY 12561. E-mail: lik@newpaltz.edu.*

Manuscript received 25 Apr. 2014; revised 21 Aug. 2015; accepted 23 Aug. 2015. Date of publication 1 Sept. 2015; date of current version 15 June 2016. Recommended for acceptance by J. L. Träff.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2015.2475763

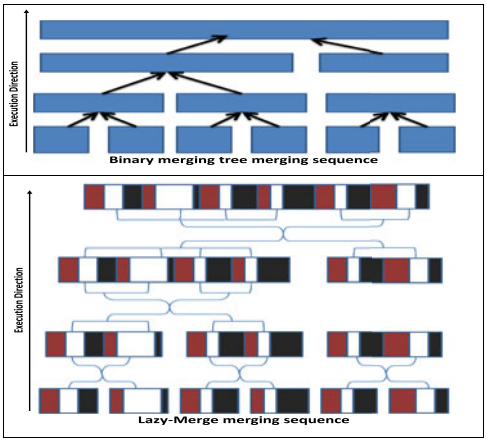


Fig. 1. Binary merging tree versus Lazy-Merge.

sizes that are powers of two. The time complexity of [10], as an example of a merging network, to merge k segments is $O(\log(k) \times (n/p) \times \log(n))$.

The parallel in-place merging is a much less addressed problem due to its impracticality to maintain the in-place constraint on networks of computational nodes. The literature includes only a limited number of research studies addressing parallel in-place merging.

For the sake of performance enhancement and cost cutting, multicore processors with shared memory architectures have become ubiquitous on computing devices, e.g., GPUs and SMP. Considering these new architectures, providing a practical and efficient algorithm for the merging problem on these architectures becomes increasingly important, particularly in the case of limited memory or shared memory. In general, there is a demand for space- and time-efficient parallel in-place algorithms.

In this paper, the research is motivated by the fact that the conflict of in-place and parallel algorithms has faded as new parallel architectures have emerged, i.e., multicore processors with shared memory. Herein, we propose Lazy-Merge, a novel implementation of sequential in-place k-way merging algorithms, that can be utilized in their parallel counterparts. Lazy-Merge utilizes indexing to merge non-contiguous segments to minimize the number of moves. The name "Lazy-Merge" comes from the fact that the implementation does not complete the merging process to the end. The final merged segment of the k input segments is not fully ordered, which can be accessed as a fully ordered list with the assist of an index of indirect pointer array. Lazy-Merge reshapes the task of k-way merging into t smaller k-way merging tasks. Each of the t tasks is addressed independently, and all the elements of a certain task are less than all of the elements of the following tasks and greater than all of the elements of the previous tasks. Lazy-Merge implementation is advantageous over the standard merging approach as shown in Section 1 of the supplemental file, available in the online supplemental material, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2015.2475763.

In the proposed work, we combine indexing with normal merging to accelerate the in-place merging process. Though this approach has a higher level of parallelism, it comes at the cost of increased access time of the final merged segment. The time complexity is, in the worst case, $O(\log k)$ sequentially and constant time in parallel, where k is the number of input segments.

The rest of the paper is organized as follows. In Section 2, we provide a modern survey of the previous work for both sequential and parallel merging algorithms. Section 3 explains the proposed parallel implementation components, accompanied by their complexity analysis. The experimental setup and performance evaluation of the proposed implementation are presented and discussed in Section 4. Finally, Section 5 concludes with the contributions of this work.

2 Related Work

This discussion of algorithms tackling the merging problem is based on the three properties and lower bounds mentioned in Section 1. In the following, we classify the discussion of these algorithms based on the utilized approach. Throughout the text, we assume that n and m are the sizes of the two segments to be merged, $m \leq n$, and that p is the number of processors.

2.1 Naive Approach

The basic sequential algorithm for merging ordered segments is stable, simple, but not-in-place. In case the input are two segments of n and m lengths, the output is placed in a new segment of length $n + m$. This method's time and space complexity are both linear. Thus, it is space inefficient for merging huge segments. The maximum numbers of comparisons and moves are $(n + m - 1)$ and $(n + m)$, respectively.

The intuitive parallel approach of the basic method has $O((n/p) \times \log(n))$ operations and $O(n + m)$ space complexity. The intuitive solution realizes the merging problem input segments as a set of blocks each of length n/p; for each element of these blocks a binary search, that costs $O(\log(n))$, is performed in the two segments to find its rank, and finally forward to its corresponding location in the output list, which has a size of $n + m$ elements.

2.2 Internal Buffer Approach

2.2.1 Sequential Methods

To overcome the space inefficiency of the basic in-place merging algorithms, the first attempt was proposed by Kronrod in [11] using a predefined constant extra memory space. Kronrod coined and used the ideas of internal buffer and block rearrangement which yielded a linear time complexity, unstable, in-place, and complex algorithm. It is based on dividing segments of lengths n into k blocks of length almost k, where k equals $\sqrt{n}$. The internal buffer concept is based on using some blocks as a workplace to merge the input segments. Several variations were presented to improve Kronrod's algorithm. One of these algorithms is the one presented by Salowe and Steiger [12]. It is impractical due to its complex structure and non-linear time complexity, albeit it is stable. Another algorithm was presented by Tridgell and Brent [13]. They proposed an unstable, in-place and simpler algorithm.

Huang and Langston proposed a stable, in-place, and simple algorithm [9]. They reported $1.5n + o(n)$ as the maximum number of comparisons and $16.5n + o(n)$ as the maximum number of assignments.

Mannila and Ukkonen relaxed the block size of Kronrod's algorithm to be of variable-length in their proposed algorithm [14]. This algorithm is unstable, in-place, complex, and with optimal number of comparisons. As one advantage of this algorithm, it takes benefit of the *sortedness* of the original segments to improve the execution time. If the entire elements are less than or equal to the first element in the second segment, e.g., the two segments need to be concatenated, then the algorithm runs in a sublinear time equal to $O(\sqrt{n})$. As a variant based on [14], Symvonis produced a stable, in-place and simple algorithm [15]. This algorithm is linear in time, but non-optimal regarding the number of comparisons.

A further step to enhance [14] was accomplished by Geffert et al. [16], then enhanced by Chen [17], [18]. Geffert et al. proposed a stable, in-place, and complex algorithm with $5n + 12m + o(m)$ maximum number of assignments, and $m(t+1) + n/2^t + o(m)$ as the maximum number of comparisons, where $t = \lfloor \log(\frac{n}{m}) \rfloor$. Chen [17] simplified the algorithm of Geffert et al. but increased the constant of proportionality to be $6n + 7m + o(n+m)$ maximum number of assignments and $m(t+1) + n/2^t + o(n+m)$ as the maximum number of comparisons, using the same assumptions for t, n, and m as Geffert et al. algorithm. In [18] Chen reported the implementation and results of his simplified algorithm; these results indicate a poor performance of this approach.

To the best knowledge of the authors, the most practical algorithms based on the internal buffer approach was introduced by Kim and Kutzner in [19] and [20]. The former algorithm is stable, in-place, and complex; it also performs optimal number of comparisons and assignments, but it is not optimal regarding the memory space. The latter algorithm has the same properties of the former, but with simpler structure that is reflected in a better performance regarding the execution time as the experimental results shown in [20]. The former algorithm is based on [14] and the internal buffer approach; meanwhile the latter uses the ratio value of the input segment lengths to simplify the algorithm.

2.2.2 Parallel Methods

Guan and Langston extended the sequential algorithm proposed in [9] and proposed the first time-space optimal parallel in-place algorithm in [7]. They defined the time-space optimality as the ability of the algorithm to achieve optimal speedup and used a constant amount of extra space per processor, regardless of the number of processors. The proposed algorithm has time complexity of $O\left(\frac{n}{p} + \log(n)\right)$ and fixed space complexity per processor , accumulating to $O(p)$ for p processors, where $p \leq \frac{n}{logn}$. They also explained how to modify the proposed algorithm to make it stable, which makes the algorithm more complicated and increases the constants of proportionality. Throughout the remainder of this paper, we refer to this algorithm as GL-Merge algorithm, for Guan and Langston.

Katajainen et al. proposed the second time-space optimal parallel algorithm in [8]. The proposed algorithm is in-place and complex, and has performance of $O(\frac{n}{p} + \log(n))$ on EREW PRAM and $O(\frac{n}{p} + \log(\log(m)))$ on CREW PRAM, using p processors.

2.3 Splitting Approach

Dudzinski and Dydek's algorithm, introduced in [21], is the first practical algorithm not based on the internal buffer approach. They used the *divide and conquer* and *block exchanging* techniques to merge the input segments. This algorithm is the seed for the *merge_without_buffer* function included in the C++ Standard Template Libraries (STL), as reported in [19]. Dudzinski and Dydek's algorithm is stable, in-place, and simple. Its number of comparisons is $O(m \times \log(n/m+1))$, and the number of assignments is bounded by $O((m+n) \times \log(m))$ in the worst case, and $O(\log(m))$ for the extra space.

Kim and Kutzner [22] introduced a stable, in-place and simple algorithm. It performs optimally regarding the maximum number of comparisons, but it is not optimal regarding the number of assignments. Another advantage of this algorithm is that, similar to [14], it can reduce the bound of the maximum number of comparisons to $O(\log(m+n))$. The algorithm is based on the rotation and the principle of symmetric comparisons. Kim and Kutzner [23] introduced a similar algorithm based on the principle of symmetric splits in addition to rotation instead of symmetric comparisons. It is a stable, in-place, and simple algorithm. It performs optimally regarding the number of comparisons and is bounded by $O((m+n) \times \log(m))$ regarding the number of assignments. Despite its higher time complexity compared with the former algorithm, the reported execution time of [23] is less than [22]; the authors attributed this conflict to the simpler structure of [23].

2.4 Shuffling Approach

Ellis and Markov proposed a new approach to tackle the 2-way merging problem. They realized the problem as a perfect shuffle permutation. In [24], Ellis and Markov proposed a stable, in-place, and simple algorithm that performs with time complexity $O(n \times \log(\log(m)))$ on average and performs linearly for balanced inputs. Dalkilic et al. proposed another shuffle-based algorithm that is stable, in-place, and simple. In that work, the authors did not provide any analysis regarding the time complexity and the lower bounds, but the experimental results exhibited a better execution time compared with that of the algorithm proposed in [24].

2.5 Merging Network

A merging network is a mathematical model of a network. This network consists of wires and comparator modules. Each comparator can access two elements and perform the merging, through a binary comparator, by sending the element with the smaller value to one output wire and the larger to the other. It performs the merging, or sorting, task in-place.

Bitonic merge is a merging network that is used in merging bitonic sequences. A sequence is bitonic if it

monotonically increases, and then monotonically decreases. In other words, it has two sub-sequences each of opposite order direction. This network was proposed by Batcher in [10]. The main idea is to use the comparison tree model of parallel computation. Given a bitonic sequence, the two sub-sequences can be merged in r levels, where the sequence length equals 2^r. In each level, there are 2^{r-1} comparisons and 2^{r-1} moves, in the worst case.

Though the model is designed for parallel environments, its time complexity is as high as $O((n/p)\log(n))$ when the number of processors is less than the size of the input lists. Thus, the time complexity is higher than the linear time complexity reported in [7], [8]; however, merging networks require a level-wise synchronization, rather than the element-wise synchronization required by [7]. The bitonic merge network is restricted to lists whose sizes are powers of two. Herein, level-wise synchronization means that the processors are synchronized only at the beginning of each level; while, element-wise synchronization means that the processors are synchronized after each data movement, i.e., elements swamping.

2.6 Summary

To summarize, the literature has very limited contributions on the parallel 2-way in-place merging task, possibly because it is less advantageous over a network of nodes. However, the spread of multicore CPUs, GPUs and SIMD architectures increases the need for such algorithms. Moreover, the ease of using recursive 2-way merging to merge k lists may allow parallel k-way merging to be seen in a less positive light.

To the best of the authors' knowledge, the literature contains two parallel 2-way in-place merging algorithms, presented in [7] and [8], in addition to the bitonic merge [10].

3 Lazy-Merge: The Proposed Implementation

The proposed implementation consists of three parts. The first part describes the process and purpose of the input segments partitioning; this part redefines the original k-way merge into exact size t smaller k-way merge, where t is the number of partitions. The second part describes the process of merging non-contiguous segments. The final part is an algorithm to access the non-fully merged segment in the correct order.

In the following discussion, we need to differentiate two terms, segment and partition. A segment presents a list of contiguous elements within a certain range, where the initial input for the k-way merging is k ordered segments. A partition is presented by a set of sub-segments; each segment contributes at most one sub-segment. Thus, the partition elements are in ranges, i.e., sub-segments, and these ranges are scattered, i.e., non-contiguous, over the k segments. In the lower part of Fig. 1, we can observe that we have a six-way merging task. Thus, the number of segments is 6; each segment has three sub-segments in different colors. These colors present the partitions; each partition has a different color, i.e., red, white and black. The elements of each partition are divided over scatted ranges, i.e., sub-segments.

3.1 Overview

Lazy-Merge proposes a novel implementation for the sequential in-place merging of k ordered lists. The implementation is based on virtually merging non-contiguous segments as though they are physically contiguous. This in-place merging task can be accomplished using an indirect pointer array to avoid complex data movement required to make the merge segments contiguous. Lazy-Merge can be used to speed up range-partitioning schemes for parallel in-place merging.

Lazy-Merge splits the ordinary binary merging tree, i.e., it divides the binary merging tree into t equal-sized smaller binary merging trees to reduce the number of moves. The t equal-sized smaller binary merging trees can use a known in-place algorithm, whether sequential or parallel, to achieve the k-way merging task, excluding bitonic merge. In Lazy-Merge, the partition contains sub-segments of arbitrary sizes when the total number of these sub-segments is known. Thus, bitonic merge cannot be used in Lazy-Merge due to the arbitrary size of its sub-segments.

Fig. 1 shows the merging sequence of the binary merging tree of in-place merging at the upper part, and depicts the Lazy-Merge merging sequence at the lower part.

In the binary merging tree with the sequential 2-way merging algorithm, the number of parallel merging tasks is halved after each level traversal. At the bottom level of the tree, there are three independent merging tasks, and at the top level, the root, there is only one merging task. In contrast, the Lazy-Merge algorithm divides each segment into three partitions. Thus, in the bottom level, the number of independent merging tasks is 9, and the number of tasks at the top level is three. As the number of partitions increases, the number of independent merges of the binary merging trees increases. Fig. 1 shows the merging path for only the second partition, which contains six sub-segments. It is noteworthy that the partitions are of equal size and that each partition merges the sub-segment on-distance in its location (by on-distance we mean that Lazy-Merge does not swap the sub-segments in such a way that each partition's sub-segments become contiguous).

3.2 Part I: Load Balanced Partitioning

The first step to realize the initial k-way merging problem as t smaller k-way merging tasks is the partitioning. The partitions should be of equal sizes. In Lazy-Merge, this task is accomplished by the algorithm proposed in [25]. The algorithm selects the mth smallest element of a set of k ordered segments which is a variation of the one proposed in [26]. This algorithm presents a partition as a vector of indexes; each component of this vector shows the index of where this partition ends for each segment. Though each partition maintains only the ending indexes of the k segments, the starting indexes can be obtained from the previous partition. Thus, the maximum size of this vector, which represents the partition, is k; in the case where the partition elements are spread over the k segments, and the minimum size is 1, the entire set of elements exist in one segment.

To form one partition out of the k segments, the time complexity is $O(k \log(f(n/k)))$, where $f, 0 < f \leq 1$, is a fraction that determines the length of a partition, and n is the sum of

the lengths of the input segments. For example, when dividing three segments with a total length of n into 4 partitions, the first partition should include the smallest $\lceil fn \rceil$ elements, where $f = 0.25$. Similarly, the second partition should include the smallest $\lceil 0.5n \rceil$ elements.

This algorithm is implemented as part of [27] MCSTL, a multicore version of STL. In this vein, another algorithm is proposed [28] that has almost the same time complexity. The algorithms' performance is almost the same, but we use the latter for its simple structure. The intuitive consequential step is to perform a block rearrangement to make the sub-segments of the partition contiguous, but Lazy-Merge starts the merging process as a non-contiguous sub-segment; thus, it is called Lazy-Merge.

Definition 1. *Let set $L = \{L_1, L_2, \ldots, L_k\}$ represent the k input segments, where L_i is a sorted segment. Let set $P = \{P_1, P_2, \ldots, P_t\}$ represent the t partitions, and $P_i = \{L_{1(u_i,v_i)}, L_{2(u_i,v_i)}, \ldots, L_{k(u_i,v_i)}\}$, where $L_{j(u_i,v_i)}$ represents a sub-segment of segment j, this sub-segment belongs to partition i, and this sub-segment's starting and ending indexes are u_i and v_i, respectively.*

From Definition 1, we can conclude that all elements of P_i are less than all elements of $P_{(i+1)}$. Lazy-Merge assumes that each P_i is an independent merging task, where $L_{1(u_i,v_i)}$, $L_{2(u_i,v_i)}, ..., L_{k(u_i,v_i)}$ are non-contiguous sub-segments of the k input segments.

3.2.1 Analysis of Number of Moves in The Binary Merging Tree

Without loss of generality, we consider the k segments to be of exact size. In the following, we use the term number of moves to represent the number of swaps, i.e., exchanges, because the in-place merging algorithm's data movement is performed as a sequence of swaps. Additionally, for k segments, the binary merging tree consists of $\lceil \log k \rceil$ levels. At the top of the tree, level 1, there is a single merge of two segments each with size $n/2$, and at the bottom of the tree, i.e., level $\lceil \log k \rceil$, there are $\lfloor k/2 \rfloor$ pair merges. Thus, the $\lceil \log k \rceil$ levels have $k - 1$ pair merges. Intuitively, as the number of segments k increases, the number of moves increases. Lazy-Merge traverses a complete level of the binary merging tree before starting the next level, level-by-level approach.

3.3 Part II: Indexing Non-Contiguous Sub-Segments

Generally, for merging algorithms, it is required to provide four inputs, the starting and ending indexes of the first and second segments. We denote these four variables in this discussion as *start1*, *last1*, *start2*, and *last2*, respectively. Of note, if the two segments are contiguous, then the value of *start2* equals that of $last1 + 1$. Definition 1 outlines that the merging task includes non-contiguous sub-segments. Thus, the direct usage of any in-place merging algorithm is incorrect because all previous in-place merging algorithms assume contiguous segments to be merged, as discussed in Section 2.

Definition 2. *Let $P_i = \{L_{1(u_i,v_i)}, L_{2(u_i,v_i)}, \ldots, L_{k(u_i,v_i)}\}$, represent a partition's start and end indexes of the sub-segments composing this partition. Let Set $V_i = \{(0, size(L_{1(u_i,u_i)})), (\sum_{r=1}^{1} size(L_{r(u_i,u_i)})) + 1, (\sum_{r=1}^{2} size(L_{r(u_i,u_i)})), (\sum_{r=1}^{2} size(L_{r(u_i,u_i)})) + 1, (\sum_{r=1}^{3} size(L_{r(u_i,u_i)})), ..., (\sum_{r=1}^{k-1} size\ (L_{r(u_i,u_i)})) + 1, (\sum_{r=1}^{k} size(L_{r(u_i,u_i)}))\}$ represent the virtual contiguous indexes for the sub-segments of partition P_i, where the first index is zero, and* size *indicates the sub-segment size.*

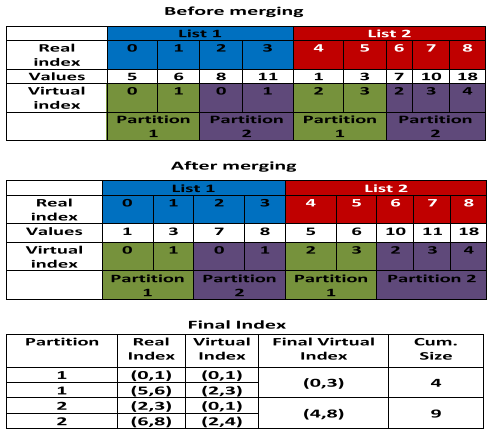


Fig. 2. Real and virtual indexes.

The proposed solution for this conflict is to maintain indexes for each partition, as shown in Definition 2. The approach is to define a set of virtual contiguous ranges. Thus, each sub-segment range belonging to P_i has a corresponding virtual range in set V_i, where set V_i contains contiguous ranges that start from zero. Fig. 2 contrives a simple example to illustrate the idea. The figure contains two segments in different colors; each is divided into two partitions. Fig. 2 shows the segments before merging, the segments after merging and illustrates both the real and virtual indexes in look-up table form. This auxiliary look-up table is used to access the not fully ordered final mergered list as a fully ordered list. In the following, we refer to this look-up table as *rosetta index*. In Fig. 2, P_i is represented by the "Real Index" column, and V_i is represented by the "Virtual Index" column.

3.3.1 The Algorithm

Algorithm 1 uses the following variables:

1) An integer n; it is the size of the k input lists.
2) A look-up table similar to the last table of Fig. 2. $V_{(partitionID)}$ and $P_{(partitionID)}$ are retrieved from this table.
3) An integer $virtualIndex$; it represents the virtual index that should be converted to the real one.
4) An integer t; it represents the number of partitions used by Lazy-Merge.

To convert a virtual index into a real one, the algorithm should address two pieces of information, the partition number and the sub-segment number of this partition, P_i, from Definitions 1 and 2.

The calculation of the partition number is straightforward. Without loss of generality, the partition number of any virtual index can be calculated by rounding down the result of dividing this index by the partition size, because the partitions have equal sizes. For example, as in Fig. 2, to obtain the partition of virtual index 7, we can calculate

$\lfloor 7/4 \rfloor$, where 4 is the partition size, to yield 1 for partition number 1, where the partition numbering starts from zero.

Consequently, the partition virtual index vector V_i should be searched to locate the correct virtual range. Because vector V_i has at most k ordered ranges, refer to the third column as the "virtual index" in Fig. 2, then, the search can be performed by a binary search algorithm. Finally, the real index can be found by accessing the corresponding range in P_i using the look-up table.

3.3.2 Complexity Analysis

From Algorithm 1, Step 1 is a constant time operation. Step 2 is a binary search task; the size of the array to be searched is at most k, where k is the number of segments. Thus, this step is of $O(\log(k))$ time complexity. Additionally, Step 3 is of constant time because it is an assignment step. The steps of Algorithm 1 have cumulative time complexity of $O(\log(k))$.

Algorithm 1. convertVirtualIndexToReal

int convertVirtualIndexToReal**(**n, virtualIndex**)**
(1) partitionID = $virtualIndex/\lfloor n/t \rfloor$;
(2) subsegmentID = binarySearch($V_{(\text{partitionID})}$, virtualIndex)
(3) realIndex = (virtualIndex - $V_{(\text{partitionID})}$[subsegmentID]. start) + $P_{(\text{partitionID})}$[subsegmentID].start
(4) **return** realIndex

Theoretically, Algorithm 1's time complexity can be reduced to constant time; if we have k ranges in set V_i and k available processors, then the binary search can be replaced by assigning one range of V_i to one processor and checking the entire ranges at once. Then, only one processor can find the correct range because the searched virtual index is located only in one virtual range. Under this assumption, Algorithm 1 can be completed in constant time. Practically, merging the segments is done once, but accessing the merged segments is a frequent event. Thus, using the parallel binary search is accepted to merge segments; meanwhile, for accessing elements of the merged segment, parallel binary search should be used when the number of cores/processors is larger than the number of segments, i.e., when using GPU devices.

On the other hand, accessing m successive elements of the merged list resulted from Lazy-Merge requires time complexity of $O(m)$, which exactly equals the standard merging format. In order to access m successive elements, Lazy-Merge needs to locate the partition, and then the range of the first element of the m elements, and the partition and range of the last element of the m elements. Then, the remaining $m-2$ elements are located in the ranges between these starting and ending ranges. Thus, the cost of accessing m successive elements includes two elements each with access time of cost $O(\log k)$, and accessing the remaining $m-2$ elements at cost of $O(1)$ for each element. That sums up to $O(m)$ which is essentially the same as the ordinary merging algorithms. A detailed example is presented as the last example of Section 4.3.1.

3.4 Part III: Merging Non-Contiguous Segments

The final part of Lazy-Merge is to merge non-contiguous segments, which are actually sub-segments. Referring to Fig. 2, the used in-place merging algorithm can work on the contiguous virtual ranges from column 3 as input. The only required modification is to replace any access of the list elements in the algorithm with a call for a routine that converts the virtual index to the real one; this routine should be the implementation of Algorithm 1.

To further clarify this idea, a modification of a sequential in-place merging algorithm is presented in Algorithm 2. We select the simplest merging algorithm, *SplitMerge* proposed in [23]. Algorithm 2 shows the modified sequential version of *SplitMerge*. Comparing the original and the modified algorithms, one can notice that line 6 is the only modified line because this line accesses the array a;thus, instead of using the simple comparison $a[m] \leq a[m']$, we modify the two indexes m and m' by mapping the virtual index, which is contiguous, to the real index.

Algorithm 2. Modified SplitMerge

SplitMerge(a, f1, f2, last)
u is in $a[f1 : f2 - distance]$, v is in $a[f2 : last - 1]$
(1) **If** $f1 \geq f2$ or $f2 \geq last$ **Then Return**
(2) $l = f1;\ r = f2;\ l' = f2;\ r' = last$
(3) **Repeat**
(4) **If** $l < r$ **Then** $m = (l + r)/2$
(5) **If** $l' < r'$ **Then** $m' = (l + r)/2$
(6) **If** $a[convertVirtualIndexToReal(m, a.size)] \leq a[(convertVirtualIndexToReal(m', a.size)]$ **Then** $l = m + 1;\ r' = m'$
(7) **Else** $l' = m' + 1;\ r = m$
(8) **Until** $l \geq r$ and $l' \geq r'$
(9) rotate(a, r, $f2$, l')
(10) SplitMerge(a, $f1,\ r,\ r + r' - f2$)
(11) SplitMerge(a, $l + l' - f2,\ l',\ last$)

A note on stability. Lazy-Merge's stability depends on the utilized partitioning algorithm and an in-place merging algorithm. The current Lazy-Merge implementation is unstable due to the partitioning algorithm. The methods proposed in [25] and [28] establish partitions without considering the stability property. In the case where there is another partitioning algorithm that establishes partitions that maintain stability, Lazy-Merge is able to retain the stability property of the used merging algorithm.

3.5 Complexity Analysis

In addition to the number of input segments and the number of partitions, Lazy-Merge includes the number of used threads/processors p. Thus, the analysis of Lazy-Merge includes k input segments, t partitions representing t independent merging tasks, and p threads. Lazy-Merge consists of two steps, partitioning the k input segments and merging the t partitions.

To form one partition of the k segments, the time complexity is $O(k \log(f(n/k)))$, where f, $0 < f \leq 1$, is a fraction that determines the length of a partition and n is the sum of the lengths of the input segments. Lazy-Merge partitions the input segments with $f = \{\frac{1}{t}, \frac{2}{t}, \ldots, \frac{t-1}{t}, \frac{t}{t}\}$. Thus, the partitioning time of $f = 1$ has the longest time, then the partitioning time complexity of $O(k \log(n/k))$. Moreover, each partition has to maintain a vector of at most k components (refer to Section 3.2), and thus the total space complexity for t partitions is $O(tk)$. Of note, when f equals 1, this means

there is one partition which containing k ranges. Each range of this partition represents an entire segment. Thus, while f equals 1 means there is no need to run the partitioning algorithm, but if the algorithm runs for $f = 1$, then it consumes the longest time.

In the second step, Lazy-Merge merges each partition, independently, using the partition information provided in the first step. The merging step's time and space complexities depend on the used in-place merging algorithm. Thus, if we consider the time complexity of the in-place merging algorithm to be $merge(n)$, then Lazy-Merge's time complexity should be $O((t/p)merge(n/t))$. The term (t/p) presents the number of partitions that will be handled by a single thread because Lazy-Merge divides the main merging task into t independent merging tasks, which can be addressed in parallel using p processors/threads. The used in-place merging algorithm should have a constant extra memory usage, so this latter step, in-place merging, has nothing to add to the space complexity. For example, the algorithm proposed in [7] has $O(n)$ time complexity, and its space complexity is $O(1)$. Using Lazy-Merge, the time complexity becomes $O((t/p)(n/t + \log(n/t)))$, and the space complexity becomes $O(tk)$ for *rosetta index*. Intuitively, each thread should maintain $O((t/p)k)$ extra memory or $O(k)$ extra memory, when $t = p$.

Lazy-Merge's rosetta index is constructed for each partition, thread. Each partition has up to $2k$ ranges; k virtual ranges, and their corresponding k real ranges. Thus, the exact size of the rosetta index is $4 \cdot k \cdot typesize \cdot t$ bytes, where 4 is for the boundaries of the virtual and real ranges of each sub-segment, the $typesize$ represents the number of bytes used for the number describes a single boundary value, and t represents the number of partitions, threads.

The bitonic merge and the GL-Merge use an auxiliary list of size p for the swapping purpose; each processor/threads use $O(1)$ extra space. The algorithm is in-place when the used extra space is no more than $O(\log^2(n))$ bits [1] (Chapter 5, Section 5, Exercise 3), and as proposed in [23] and [24]. Thus, Lazy-Merge is in-place as $k \leq \log^2(n)$ per processor, where n is the sum of the lengths of the k input segments and $t = p$.

Finally, because the merged elements are not in the standard format, accessing an element requires a binary search within the corresponding partition's k ranges. Thus, the access time of an element of the final merged list is $O(\log(k))$; that time can be constant if k processors/threads are used, as detailed in Section 3.3.2. In Section 1 of the supplemental file, available in the online supplemental material, we explained an application that gains a better execution time as it benefits from this non-standard format of Lazy-Merge's output.

To summarize, Lazy-Merge's time complexity is $O(k \log(n/k)) + (t/p)merge(n/t))$, and its space complexity is $O(tk)$. Without loss of generality, if we consider the number of threads to be equal to the number of partitions, $t = p$, then Lazy-Merge's time and space complexities are $O(k \log(n/k)) + merge(n/p))$ and $O(pk)$, respectively. Each thread maintains a vector of ranges of size $O(k)$, so the access time of the final merged list is $O(\log(k))$ using one processor, that time can be constant if k processors/threads are used.

In the following, for the bitonic merge and the GL-Merge algorithms, we will use the term thread. For Lazy-Merge, we will use the terms *thread* and *partition* interchangeably, because we assume that the number of partitions equals the number of threads.

4 Performance Evaluation

4.1 Experimental Setup

The experiments are performed on a computer with two 2.3 GHz AMD Opteron 6134 processors, each containing 8 cores, and 8 GB of RAM. The OS utilized is 64-bit Linux. All algorithms are written in the C++ programming language. We used the standard OpenMP threads library [29].

We proposed using two different datasets of normally distributed integer numbers. Because the bitonic merge network is restricted to lists whose sizes are powers of two, the input list total size should be power of two. The first dataset list sizes are in the tens of thousands. It has five input lists; they vary from 2^{13} to 2^{17}, and we call it dataset-1. The other dataset list sizes are in millions. It has has seven input lists; they vary from 2^{20} to 2^{26}, and we call it dataset-2.

For each dataset, each list is tested with different k values, where $k = 2^r$ and $1 \leq r \leq 7$. Each variation of each list is tested using a different number of partitions t, where $t = 2^u$ and $0 \leq u \leq 7$; for the entire suite of tests, we supposed that the number of partitions, t, was equal to the number of threads, p. Thus, each list belonging to the two datasets will be tested 7×8 times, where 7 and 8 represent the number of segments and threads, respectively.

4.2 Selecting the Merging Algorithm

Section 2 shows that there are four approaches for sequential in-place merging. To test the proposed algorithm, we need to select a sequential algorithm for the 2-way merging. We excluded the internal buffer-based approach due to its high number of moves, as discussed in Section 2. Additionally, we excluded the bitonic merge due to the powers of 2 size restriction and the fact that Lazy-Merge divides the segments into arbitrary sized sub-segments. We compared the remaining two approaches to select the one with better performance. We have compared the *ShuffleMerge* algorithm in [24] as a shuffle-based algorithm and the *SplitMerge* algorithm in [23] as a split-based algorithm, which is considered the fastest in-place merging algorithm, as reported in [23].

To enhance the execution time of the *ShuffleMerge* algorithm, we replaced the original shuffling algorithm with a new faster shuffling algorithm that is proposed in [30]. The comparison experiments are presented in Section 2 of the supplemental file, available online. The experimental results show that the *SplitMerge* algorithm has an overall better performance than *ShuffleMerge*, especially for large list sizes and partially interlaced lists.

4.3 Implementation Details

To evaluate the proposed algorithm, we used *SplitMerge* [23] as a sequential merging routine of Lazy-Merge. Additionally, we selected the bitonic merge and the GL-Merge algorithm [7], because the former is representative of merging networks and the latter is the most recent parallel space-time optimal in-place merging algorithm. We considered

the size restriction of the bitonic merge for the entire utilized datasets. In the GL-Merge algorithm, we used *SplitMerge* as the merging routine instead of [9], which is used in the original GL-Merge algorithm. We did so to make the merging routines uniform for both the Lazy-Merge and the GL-Merge algorithms. In the following, we discuss the implementation details of the algorithms under comparison.

4.3.1 Lazy-Merge

For Lazy-Merge, the algorithm assigns a single core/thread to merge the ranges of one partition, $t = p$. Thus, each thread performs a 2-way merging binary tree for the non-contiguous sub-segments of the corresponding partition. It handles different smaller binary merging trees.

Using a binary merging tree with 2-way in-place merging, the results in the final merged list can be accessed in constant time because the final merged list is in the standard format. In contrast, Lazy-Merge handles non-contiguous segments; it has a higher cache miss ratio because the binary merging tree accesses contiguous elements. A larger CPU cache results in better Lazy-Merge performance. In the following, we discuss two code optimization techniques to speed up the element retrieval. These techniques can be used after the partitioning step because the indexes table is data dependent and known only at run time. The techniques are used to merge the non-contiguous ranges, and after Lazy-Merge completes the merging task, to retrieve the merged elements, which are in the non-standard merging format.

Lazy-Merge has a $O(\log(k))$ access time, where k is the number of segments. To enhance Lazy-Merge's performance, we suggest using a cache structure for the range indexes, not the elements. Thus, instead of converting the virtual index to the real index for each element access, we store the virtual and real ranges of the last accessed virtual index; these ranges are used for the next retrieval. For the next retrieval, the virtual index is checked against the fetched range; if it belongs to the fetched range, then the access time is constant; otherwise, it costs $O(\log(k))$. This process can be extended to fetch a number of successive and preceding ranges, which should reduce the search space of the original one, particularly for large numbers of segments. In our implementation, we use a cache of size 1, storing one virtual range and its corresponding real range.

We elaborate on the concept through the following example. We are given the indexes table with virtual ranges $V = \{(1,5),(6,7),(8,13),(14,16)\}$ and its corresponding real ranges $R = \{(12,16),(7,8),(9,11),(1,6)\}$. We use the cache structure technique and iterate through a loop that accesses elements with virtual indexes from 5 to 8. Because index 5 is the first element, the cache is empty, and a binary search in the indexes table is performed to find the virtual range $(1,5)$ and the real range $(12,16)$. For the next index 6, the cached range is not the corresponding one, then another fetch operation is performed. For the next index 6, the cache already has the corresponding ranges, and the access time is constant. For index 8, its corresponding ranges are not in the cache; thus, a final fetch operation is performed.

Another technique is to convert the loops of virtual indexes to loops of real indexes. The merging process contains many loops. Intuitively, these loops access contiguous elements. This should be faster because the mapping from virtual to real index is done for ranges rather than individual elements.

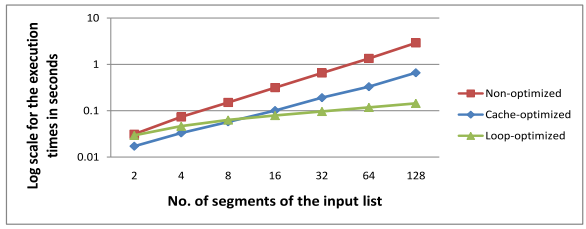


Fig. 3. Lazy-Merge's implementation techniques exec. times.

The following example illustrates this technique with the assistance of the aforementioned example. If we need to convert the virtual indexes in a loop starting from index 7 to 14, then we need to find the virtual ranges in this loop rather than converting each element in the loop. This can be done by searching only the first and last indexes of the loop. Thus, we first binary search for the first virtual index, 7, to find the virtual sub-range $(7,7)$ in the range $(6,7)$. Afterward, we binary search for the last virtual index, 14, to find it in the sub-range $(14,14)$ in the range $(14,16)$. Finding the corresponding real ranges requires constant time because the rank of the virtual range in vector V is the same as the rank of the real range in vector R. Thus, looping from virtual index 8 to 14 is accomplished by looping through the corresponding real ranges $(8,8),(9,11),(1,1)$.

Fig. 3 depicts the execution times for the two techniques against the unoptimized Lazy-Merge, converting each virtual index access to the real one. The running times in Fig. 3 are the average running times for the lists of dataset-2 using 128 partitions. This figure shows better execution times for the last mentioned technique. That will be used for the entirety of the following experiments.

4.3.2 Bitonic Merge

There are two variations of bitonic merge, the level-by-level emulation and the perfect shuffle-based variation [31]. The latter has longer running time because it includes extra time to complete the perfect shuffle effect, as discussed in [32]. The shuffle-based variation of bitonic merge should be more efficient on a shuffle network, but it has an extra overhead on multi-core/many-core architecture, i.e., multi-core CPUs.

Given bitonic sequences, bitonic merge sorts the two equal input segments by the recursive construction of a comparator network that merges any bitonic sequence. Bitonic merge completes the merging in $\log(n)$ levels; each level divides the subject segments into two equal sub-segments, and it then performs the comparison and exchange effect and recursively calls the bitonic merge for each of the two halves that belong to the same segment.

Generally, recursive functions are expensive, although some compilers optimize tail recursion (not all do so). Thus, we implemented the bitonic merge as an iterative function. Moreover, we parallelized the loops for comparing and conditionally exchanging elements to speed up the execution

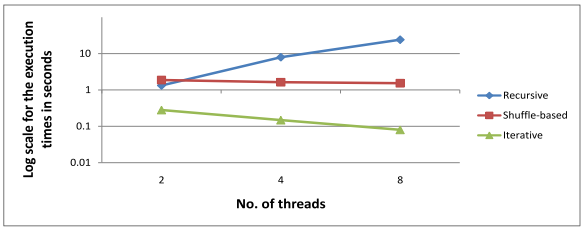


Fig. 4. Bitonic merge different implementations exec. times.

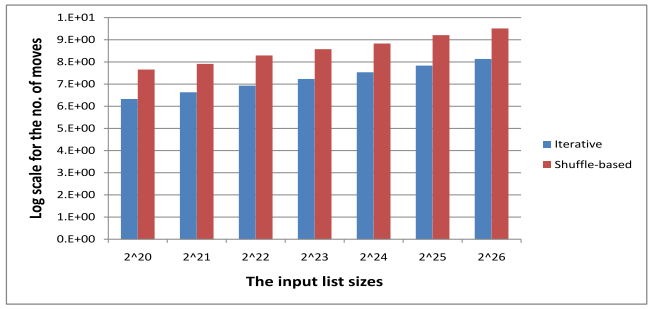


Fig. 5. Bitonic merge no. of moves for different implementations.

time of bitonic merge. Herein, in Figs. 4 and 5, we provide a comparison between the three implementations, the original recursive algorithm, our optimized iterative version, and the shuffle-based variation. The shuffle-based bitonic implementation uses the in-place perfect shuffle algorithm which is proposed in [30]. We implemented a parallel version of this shuffling algorithm; we parallelized the second step, right cyclic shift by a certain distance. Additionally, we convert the recursive version to an iterative one.

We used a logarithmic scale to depict the execution times in Fig. 4, for the first list in dataset-2, and we used a two segments for the entire test. In Fig. 4, the recursive implementation execution time increases as the number of threads increases. This is because each recursive call has to start and manage new p threads. In contrast, our iterative implementation start and manage the p threads $\log n$ times, once for each level. Finally, the shuffle-based bitonic implementation has a slight performance enhancement as the number of threads increases due to the shuffle overhead. The shuffle algorithm has a single parallel step while the remaining steps still sequential. Thus, the iterative implementation of bitonic merge scales better than the other two implementations as the number of threads increases.

Similarly, Fig. 5 depicts the number of moves for lists of dataset-2 with two segments, changing the number of threads has no effect on the number of moves for bitonic merge. The bitonic recursive and iterative implementations have the same number of moves. Thus, we include the iterative version only in Fig. 5. Fig. 5 shows a massive number of moves difference. The huge number of moves of the shuffle-based bitonic merge is due to adding the moves which are used by the in-place perfect shuffle algorithm.

4.3.3 GL-Merge

Algorithm 3 lists the main steps of the GL-Merge algorithm. Using a list of two segments and a total size of n, Step 1 divides the n elements into p equal-sized blocks, where p represents the number of threads. Step 2 sorts the tails list of size p, using bitonic sort; this sorting determines the destination block for the current blocks. Step 3 forwards an element from each block at a time to another block; thread i iterates block i elements. Step 3 has a loop of n/p, block size, iterations and an auxiliary list of size p, one element for one block. In each iteration, two synchronized cycles are utilized. In the first cycle, thread i writes the current element to location i of the auxiliary list. In the second cycle, thread i reads the element j from the auxiliary list, where j represents the block index that should replace block i, where block i elements may reside in block j or any other block depending on the sorting result. In Step 4, each thread i searches block i for an element greater than the tail of block $i-1$. This step can be done using binary search. In Step 5, each block calculates how many element will be replaced by elements from the other blocks. Similar to Step 3, the p blocks exchanges elements using two synchronization cycles for each iteration in Step 6. The results of Step 6 are re-arranged blocks; each contains a range of ordered non-displaced elements and another range for the new received elements; each block independently merges these two ranges in Step 7.

Algorithm 3. GL-Merge Algorithm

1) Divide the input list into p blocks; each of size n/p.
2) Sort the blocks tails s.t. each block knows its destination block.
3) Re-arrange the blocks such that their tails are in order.
4) In each block, find an element with a value greater than the previous block tail value, the breaker.
5) Using breakers, calculate the displacement table.
6) Move blocks elements according to the displacement table.
7) Each block independently merges the range of the non-displaced elements and the range of the received elements from the other blocks.

The most expensive steps of the GL-Merge algorithm are Steps 3 and 6, which include the data movement of elements between blocks. The number of synchronization is $O(n/p)$ that should result in poor performance. The number of synchronization can be reduced by techniques like Lazy synchronization, but that come at the cost of more space. Thus, reducing the synchronization number is contraindicating with the in-place condition.

The Parallel Random Access Machine (PRAM) model neglects the synchronization and communication cost. It consists of p synchronous processors while most of the new parallel architectures contain asynchronous multiprocessors. Thus, the PRAM model, while fruitful from a theoretical perspective, is proved unrealistic [33], [34]. For example, arbitrary, sparse graphs problems have abundant theoretically optimal parallel PRAM algorithms; from practical perspective, few parallel implementations outperform the best sequential implementations [35]. Similarly, for the GL-Merge algorithm, the massive number of required synchronization makes it impractical for real parallel architectures, like CPUs and GPUs, while it is theoretically optimal for the PRAM model. A thorough comparison between

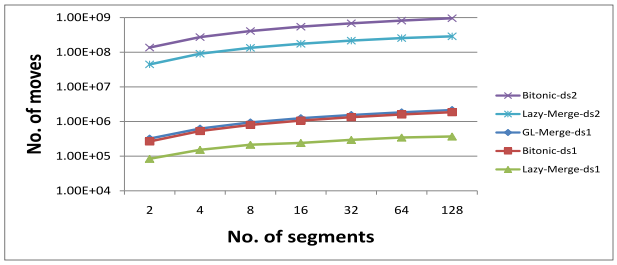


Fig. 6. Number of moves for different number of segments for the largest list of dataset-1 and dataset-2.

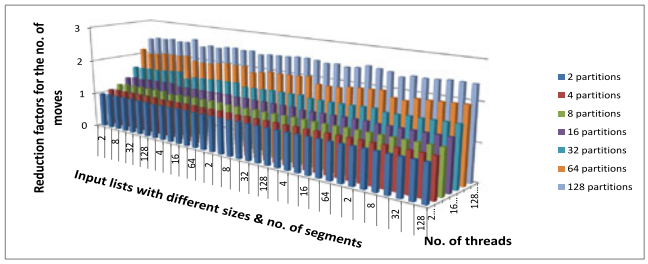


Fig. 8. GL-Merge no. of moves reduction factor for dataset-1.

Lazy-Merge and GL-Merge is presented in Section 3 of the supplemental file, available online.

4.4 Experimental Results and Discussion

4.4.1 Evaluating the Number of Moves

Fig. 6 shows the total number of moves required to merge the largest input list of dataset-1 and dataset-2 using 128 threads (smaller lists behave similarly) under a varied number of segments. The figure shows that Lazy-Merge outperforms the other two algorithms. The GL-Merge algorithm and Bitonic merge have a close total number of moves in dataset-1, thus, their lines appear very close to each other.

Additionally, it should be noted that the numbers of used segments and threads do affect the number of moves. In Section 4 of the supplemental file, available online, we provide a detailed experimental study of the effect of the number of segments and threads on the total number of moves for the three algorithms.

4.4.2 Evaluating the Number of Moves Reduction

The bitonic merge performs $n/2$ comparisons at each level of the $\log n$ levels. Thus, increasing the number of threads has no effect on the number of moves. In contrast, for the Lazy-Merge and GL-Merge algorithms, both divide the list of size n into smaller independent merging problem; that should lead to a reduction in the total number of moves. Thus, we exclude the bitonic merge from this evaluation.

Figs. 7 and 8 show the number of moves reduction factors for dataset-1 for Lazy-Merge and GL-Merge, respectively. Fig. 9 depicts the number of moves reduction factors for dataset-2 for Lazy-Merge only; the GL-Merge is excluded from the tests of dataset-2 due to its massive running time. Generally, we can notice that the reduction factor of the total number of moves decreases as the input list sizes increases. Meanwhile, increasing the number of threads/partitions increases the reduction factor.

4.4.3 Evaluating the Merging Running Times

Fig. 10 shows the running times, in seconds, of the partitioning algorithm for dataset-2. We used lists with fixed 128 segments for this experiment. The running times of this figure represent the partitioning time for $f = 1$, which takes the longest time because the partitioning time increases as f increases. The figure shows an expected increase in the partitioning time as the number of segments and/or the input list size increases.

Fig. 11 depicts the execution times for the largest list of dataset-1. This input list is a representative list, because including the entire lists of dataset-1 will make the figure complex. Fig. 11 shows a similar behavior for the three algorithms. The three algorithms' execution times decrease till reaching a certain number of threads; afterward, the execution times start to increase, because the overhead of the larger number of threads is more than the reduced merging task assigned to each thread. Section 5 of the supplemental file, available online, includes the full list of execution times for the entire experiments of the three algorithms.

Fig. 12 depicts the execution times for the largest list of dataset-2. The larger sizes of the dataset-2 lists allow the two algorithms to scale well.

Figs. 13 and 14 show the speedups of Lazy-Merge against a binary merging tree with 2-way in-place merging tasks using the *SplitMerge* algorithm. We can notice that the average speedups for the series "128 partitions" in Fig. 14 is higher than the corresponding average speedups in Fig. 13, while series "8 partitions" has the opposite behavior.

Generally, the execution time of Lazy-Merge varies depending on three factors, the total size of the input segments, the number of segments, and the number of partitions/threads. We relate this decline of speedups of some

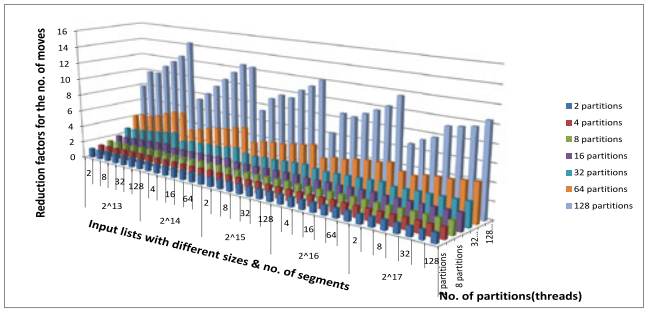


Fig. 7. Lazy-Merge no. of moves reduction factor for dataset-1.

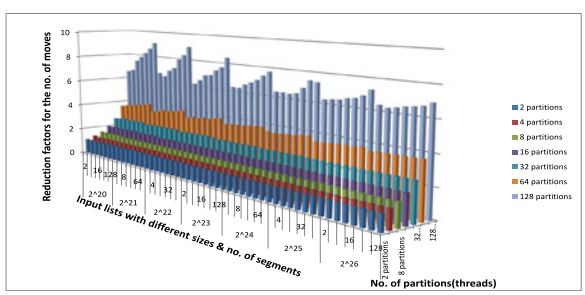


Fig. 9. Lazy-Merge no. of moves reduction factor for dataset-2.

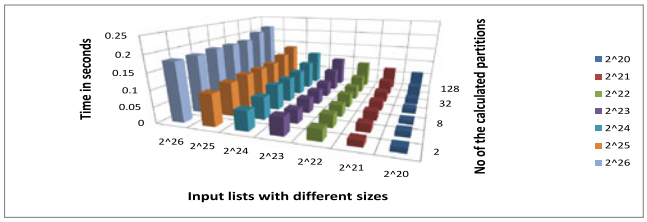
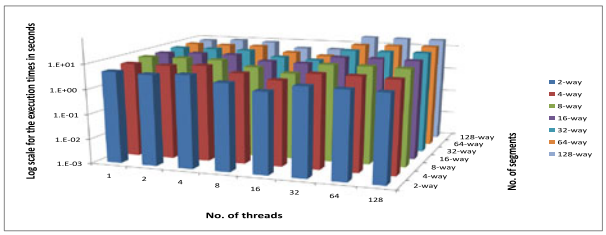
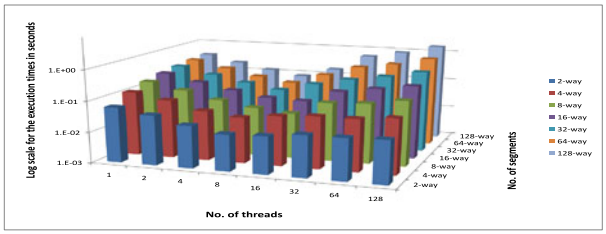


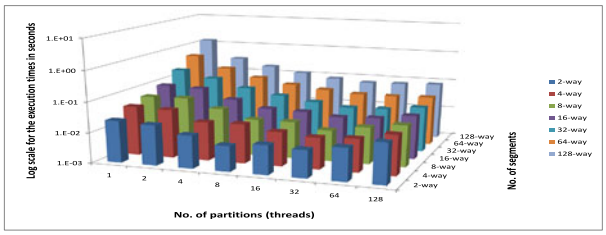
Fig. 10. Data partitioning timing for dataset-2.



(a) GL-Merge algorithm

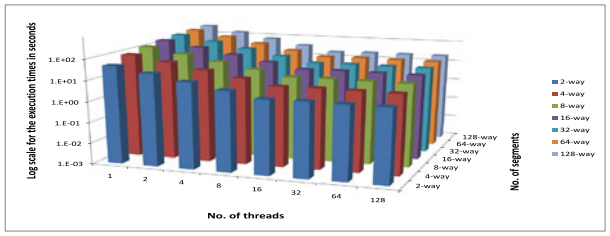


(b) Bitonic algorithm

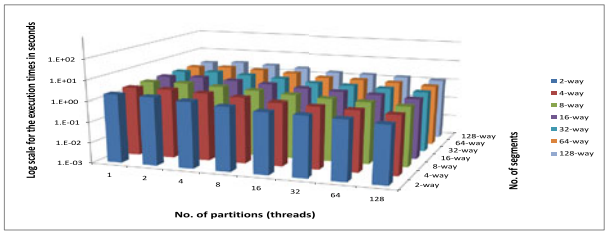


(c) Lazy-Merge

Fig. 11. Execution times for dataset-1.



(a) Bitonic algorithm



(b) Lazy-Merge

Fig. 12. Execution times for dataset-2.

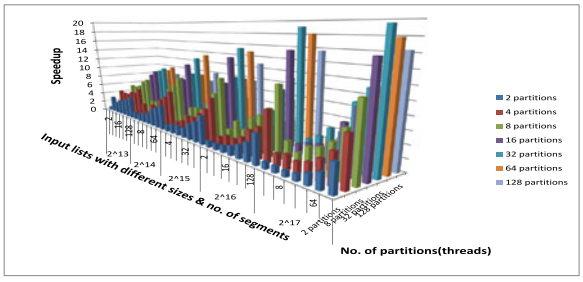


Fig. 13. Lazy-Merge's speedups for dataset-1.

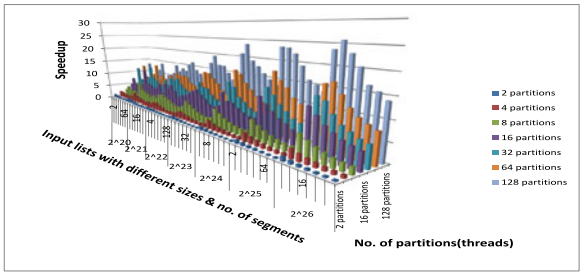


Fig. 14. Lazy-Merge's speedups for dataset-2.

series, i.e., "8 partitions", to the overhead of converting the virtual index to real index. Apparently, as the partition size increases the conversion overhead increases. Thus, the number of partitions should be relative to input lists size.

Despite this decline of the speedups, in dataset-2, Lazy-Merge outperforms the bitonic merge, regardless the number of segments or the number of partitions/threads. For example, under a list of size 2^{26} with two segments and using two and 128 threads/partitions, Lazy-Merge is faster than bitonic merge by 3.02 and 8.28 times, respectively. Using the same list with 128 segments and two and 128 threads/partitions, Lazy-Merge is faster than bitonic merge by 2.65 and 15.54 times, respectively.

For the entire execution times, Lazy-Merge outperforms the bitonic merge and GL-Merge algorithms. For example, for the largest list of dataset-2 and under 128 threads, Lazy-Merge outperforms bitonic merge by an order of magnitude. For the largest list of datatset-1, Lazy-Merge is faster than GL-Merge by at least two order of magnitude using two or more threads, regardless the number of segments.

4.4.4 *Evaluating Elements Access Time*

To evaluate the access time, we test both access types discussed in Section 3.3.2. The experiments include accessing a random element from the largest list of dataset-1 with varied number of segments, 2 to 128, and single thread to access the element(s). This process is repeated 1,000 times, and the reported times are the average times. For single element access, the bitonic merge has an average access time of 2.5^{-8} second, regardless the number of segments. For Lazy-Merge, the single element access time for two merged segments is 5.2^{-7} second; this access time slightly increases till it reaches 6.4^{-7} second for 128 merged segments. Because the single element access time

of Lazy-Merge is $O(\log k)$, the access time increases as the number of segments increases.

For accessing a set of successive elements, we used the elements in the first 30 percent of the final merged list. The reported times are the average of running the experiments three times. The average access times are 1.5^{-3} second and 1.9^{-3} second for bitonic merge and Lazy-Merge, respectively. While the bitonic merge and Lazy-Merge have the same time complexity $O(m)$ to access m successive elements, the bitonic merge slightly has better access time due to the cache locality. Bitonic merge accesses physically successive elements while Lazy-Merge accesses discrete ranges, virtually successive and physically discrete.

4.4.5 *Evaluating* Rosetta Index *Size*

Lazy-Merge utilizes *rosetta index* to map the virtual indexes into real indexes. Its size depends on the number of the segments and partitions, as discussed in Section 3.5. In *rosetta index* for each segment, each partition has two ranges, one real and one virtual. Each range boundary requires 4 bytes, integer data type size. Thus, one range costs 8 bytes, and two ranges need 16 bytes, regardless the number of elements of the input segment. Thus, the size of *rosetta index* for the largest input list of dataset-1, or dataset-2, equals 16 bytes multiplied by the number of segments per each partition.

5 Conclusions

In this paper, we consider the topic of parallel in-place merging due to its limited literature and its raised importance due to the emergence of new parallel architectures. We propose Lazy-Merge, a novel implementation for parallel k-way in-place merging algorithms. Lazy-Merge partitions the input list into t independent merging tasks; afterward, each partition merges non-contiguous ranges, with assist of an auxiliary look-up table, *rosetta index*.

Through numerous experiments, we compare Lazy-Merge with the existing algorithms on the total number of moves reduction, execution time, and extra used memory. The results indicate that the Lazy-Merge implementation outperforms the existing algorithms in the former two issues while it consumes more extra memory. Lazy-Merge's performance varies depending on three factors, the total size of the input segments, the number of segments and the number of partitions/threads. A further research could shed more light on this relationship of these factors and Lazy-Merge's performance.

To the best of our knowledge, Lazy-Merge is the fastest parallel k-way in-place merging implementation to the date. Additionally, Lazy-Merge is the first technique proposes the conception of merging non-contiguous elements; this conception can be mimicked to other computing operations where the cost of data movement is high.

Acknowledgments

The authors acknowledge the three anonymous reviewers for their detailed and helpful comments to the work, and they are thankful for Professor Peter Sanders, Karlsruhe Institute of Technology, for his discussion of the proposed algorithm. The research was partially funded by the Key Program of National Natural Science Foundation of China (Grant Nos. 61133005, 61432005), the International Science & Technology Cooperation Program of China (Grant No. 2015DFA11240), the National Natural Science Foundation of China (Grant Nos. 61370095, 61472124), and the Egyptian Ministry of Higher Education. Kenli Li is the corresponding author.

References

[1] D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, vol. 28, no. 128. Reading, MA, USA: Addison-Wesley, 1973

[2] J. r. Bang-Jensen, J. Huang, and L. Ibarra, "Recognizing and representing proper interval graphs in parallel using merging and sorting," *Discrete Appl. Math.*, vol. 155, no. 4, pp. 442–456, 2006.

[3] M. Hofmann, G. Runger, P. Gibbon, and R. Speck, "Parallel sorting algorithms for optimizing particle simulations," in *IEEE International Conf. Cluster Comput. Workshops Posters*, pp. 1–8, 2010.

[4] T. White, *Hadoop: The Definitive Guide*, 1st ed. Sebastopol, CA, USA: O'Reilly Media, Inc., 2009.

[5] G. Salton, *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*. Reading, MA, USA: Addison-Wesley, 1989.

[6] V. Geffert and J. Gajdoš, "Multiway in-place merging," in *Fundamentals of Computation Theory*. New York, NY, USA: Springer, 2009, pp. 133–144.

[7] X. Guan and M. S. Langston, "Time-space optimal parallel merging and sorting," *IEEE Trans. Comput.*, vol. 40, no. 5, pp. 596–602, May 1991.

[8] J. Katajainen, C. Levcopoulos, and O. Petersson, "Space-efficient parallel merging," in *Proc. 4th Int. PARLE Conf. Parallel Archit. Languages Eur.*, 1992, vol. 605, pp. 37–49.

[9] B.-C. Huang and M. A. Langston, "Practical in-place merging," *Commun. ACM*, vol. 31, no. 3, pp. 348–352, Mar. 1988.

[10] K. E. Batcher, "Sorting networks and their applications," in *Proc. Spring Joint Comput. Conf.*, 1968, pp. 307–314.

[11] M. A. Kronrod, "An optimal ordering algorithm without a field of operation," *Soviet Math*, vol. 10, pp. 744–746, 1969.

[12] J. Salowe and W. Steiger, "Simplified stable merging tasks," *J. Algorithms*, vol. 8, no. 4, pp. 557–571, Dec. 1987.

[13] A. Tridgell and R. P. Brent, "A general-purpose parallel sorting algorithm," *Int. J. High Speed Comput.*, vol. 7, no. 2, pp. 285–301, 1995.

[14] H. Mannila and E. Ukkonen, "A simple linear-time algorithm for in situ merging," *Inf. Process. Lett.*, vol. 18, no. 4, pp. 203–208, May 1984.

[15] A. Symvonis, "Optimal stable merging," *Comput. J.*, vol. 38, no. 8, pp. 681–690, 1995.

[16] V. Geffert, J. Katajainen, and T. Pasanen, "Asymptotically efficient in-place merging," *Theor. Comput. Sci.*, vol. 237, no. 1-2, pp. 159–181, Apr. 2000.

[17] J.-C. Chen, "Optimizing stable in-place merging," *Theor. Comput. Sci.*, vol. 302, no. 1-3, pp. 191–210, Jun. 2003.

[18] J. Chen, "A simple algorithm for in-place merging," *Inf. Process. Lett.*, vol. 98, no. 1, pp. 34–40, Apr. 2006.

[19] P.-S. Kim and A. Kutzner, "On optimal and efficient in place merging," in *Proc. SOFSEM: 32nd Conf. Current Trends Theory Practice Comput. Sci.*, vol. 3831, pp. 350–359, 2006.

[20] P. Kim and A. Kutzner, "Ratio based stable in-place merging," in *Proc. 5th Int. Conf. Theory Appl. Models Comput.*, 2008, pp. 246–257.

[21] K. Dudzinski and A. Dydek, "On a stable minimum storage merging algorithm," *Inf. Process. Lett.*, vol. 12, no. 1, pp. 5–8, 1981.

[22] P.-S. Kim and A. Kutzner, "Stable minimum storage merging by symmetric comparisons," in *Proc. 12th Annu. Eur. Symp.*, 2004, pp. 714–723.

[23] P. Kim and A. Kutzner, "A simple algorithm for stable minimum storage merging," in *Proc. SOFSEM: 33rd Conf. Current Trends Theory Practice Comput. Sci.*, 2007, pp. 347–356.

[24] J. Ellis and M. Markov, "In situ, stable merging by way of the perfect shuffle," *Comput. J.*, vol. 43, no. 1, pp. 40–53, 2000.

[25] P. J. Varman, S. D. Scheufler, B. R. Iyer, and G. R. Ricard, "Merging multiple lists on hierarchical-memory multiprocessors," *J. Parallel Distrib. Comput.*, vol. 12, no. 2, pp. 171–177, 1991.

[26] G. N. Frederickson and D. B. Johnson, "The complexity of selection and ranking in x+y and matrices with sorted columns," *J. Comput. Syst. Sci.*, vol. 24, no. 2, pp. 197–208, 1982.
[27] J. Singler, P. Sanders, and F. Putze, "MCSTL: The multi-core standard template library," in *Proc. 13th Int. Euro-Par Conf. Parallel Process.*, 2007, pp. 682–694.
[28] R. Francis, I. Mathieson, and L. Pannan, "A fast, simple algorithm to balance a parallel multiway merge," in *Proc. 5th Int. PARLE Conf. Parallel Archit. Languages Eur.*, 1993, pp. 570–581.
[29] L. Dagum and R. Menon, "OpenMP: An industry standard API for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Jan.–Mar. 1998.
[30] P. Jain, "A simple in-place algorithm for in-shuffle," *CoRR*, vol. abs/0805.1, 2008.
[31] H. S. Stone, "Parallel processing with the perfect shuffle," *IEEE Trans. Comput.*, vol. C-20, no. 2, pp. 153–161, Feb. 1971.
[32] J.-D. Lee and K. E. Batcher, "Minimizing communication in the bitonic sort," *IEEE Trans. Parallel Distrib. Syst.*,vol. 11, no. 5, pp. 459–474, May 2000.
[33] R. Dorrigiv, A. López-Ortiz, and A. Salinger, "Optimal speedup on a low-degree multi-core parallel architecture (LoPRAM)," in *Proc. 20th Annu. Symp. Parallelism Algorithms Archit.*, 2008, pp. 185–187.
[34] D. Ajwani and H. Meyerhenke, "Realistic computer models," in *Algorithm Engineering*. New York, NY, USA: Springer, 2010, pp. 194–236.
[35] G. Cong and D. A. Bader, "Techniques for designing efficient parallel graph algorithms for SMPs and multicore processors," in *Proc. 5th Int. Symp. Parallel Distrib. Process. Appl.*, 2007, pp. 137–147.

Ahmad Salah received the master's degree in CS from Ain-shams University, Cairo, Egypt, and the PhD degree in computer science from Hunan University, China. His current research interests include parallel computing, computational biology, and algorithms.

Kenli Li received the PhD degree in computer science from Huazhong University of Science and Technology, China, in 2003. He was a visiting scholar at the University of Illinois at Urbana-Champaign from 2004 to 2005. He is currently a full professor of computer science and technology at Hunan University and the deputy director in the National Supercomputing Center in Changsha. His major research areas include parallel computing, high-performance computing, and grid and cloud computing. He has published more than 130 research papers in international conferences and journals such as *IEEE Transactions on Computers*, *IEEE Transactions on Parallel and Distributed Systems*, *Journal of Parallel and Distributed Computing*, ICPP, CCGrid. He is an outstanding member of CCF. He is a member of the *IEEE* and serves on the editorial board of the *IEEE Transactions on Computers*.

Keqin Li is a SUNY distinguished professor of computer science. His current research interests include parallel computing and high-performance computing, distributed computing, energy-efficient computing and communication, heterogeneous computing systems, cloud computing, big data computing, CPU-GPU hybrid and cooperative computing, multicore computing, storage and file systems, wireless communication networks, sensor networks, peer-to-peer file sharing systems, mobile computing, service computing, Internet of things and cyber-physical systems. He has published more than 360 journal articles, book chapters, and refereed conference papers, and has received several best paper awards. He is currently or has served on the editorial boards of *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, *IEEE Transactions on Cloud Computing*, *Journal of Parallel and Distributed Computing*. He is a fellow of the IEEE.