Multiway Blockwise In-place Merging

Viliam Geffert and Jozef Gajdoš

Institute of Computer Science, P.J.Šafárik University, Faculty of Science
Jesenná 5, 041 54 Košice, Slovak Republic
viliam.geffert@upjs.sk, jozef.gajdos@upjs.sk

Abstract. *We present an algorithm for asymptotically efficient multiway blockwise in-place merging. Given an array A containing sorted subsequences $A_1, \ldots, A_k$ of respective lengths $n_1, \ldots, n_k$, where $\sum_{i=1}^{k} n_i = n$, we assume that extra $k \cdot s$ elements (so called buffer elements) are positioned at the very end of array A, and that the lengths $n_1, \ldots, n_k$ are positive integer multiples of some parameter s (i.e., multiples of a given block of length s). The number of input sequences k is a fixed constant parameter, not dependent on the lengths of input sequences. Then our algorithm merges the subsequences $A_1, \ldots, A_k$ into a single sorted sequence, performing $\Theta(\log k \cdot n) + O((n/s)^2) + O(s \cdot \log s)$ element comparisons and $3 \cdot n + O(s \cdot \log s)$ element moves.*[1]
Then, for $s = \lceil n^{2/3}/(\log n)^{1/3} \rceil$, this gives an algorithm performing $\Theta(\log k \cdot n) + O((n \cdot \log n)^{2/3})$ comparisons and $3 \cdot n + O((n \cdot \log n)^{2/3})$ moves. That is, our algorithm runs in linear time, with an asymptotically optimal number of comparisons and with the number of moves independent on the number of input sequences. Moreover, our algorithm is "almost in-place", it requires only k extra blocks of size $s = o(n)$.

1 Introduction

Given an array $A[1..n]$ consisting of sorted subsequences $A_1, \ldots, A_k$ each containing $n_1, \ldots, n_k$ elements respectively, where $\sum_{i=1}^{k} n_i = n$, the *classical multiway in-place merging* problem is to rearrange these elements to form a single sorted sequence of n elements, assuming that only one extra storage location (in addition to the array A) is available for storing elements. To store array indices, counters, etc. only $O(1)$ storage locations are available. The efficiency of a merging algorithm is given by two quantities: the number of pairwise element comparisons and the number of element moves carried out in the worst case, both expressed as a function of n. In merging, these are the only operations permitted for elements.

In this paper we study the computational complexity of the multiway *blockwise* in-place merging problem. More precisely, we assume that the entire array A is divided into blocks of equal size s, and that k extra blocks of size s are positioned at the very end of array A. Moreover, the lengths $n_1, \ldots, n_k$ of input sequences are positive integer multiples of s, and hence, there is always a block boundary between the last element of A_i and the first element of A_{i+1}, for each $i \in 1, \ldots, k-1$. We shall also assume, that before the merging starts, blocks can be mixed up quite arbitrarily, so we no longer know the original membership of blocks in the input sequences $A_1, \ldots, A_k$.

So far, the problem has been resolved for two-way merging, i.e., for $k = 2$ [4]. This algorithm uses $2n + o(n)$ comparisons, $3n + o(n)$ element moves and $O(1)$ extra locations for storing elements, in the worst case. Thus, by repeated application of this algorithm, we could carry out k-way merging in linear time, for arbitrary $k \geq 2$. However, implemented this way, the k-way merging would perform $3 \cdot \lceil \log k \rceil \cdot n + o(n)$ element moves and $2 \cdot \lceil \log k \rceil \cdot n + o(n)$ element comparisons. We shall show that the number of moves *does not depend on* k, if the lengths $n_1, \ldots, n_k$ are integer multiples of the block size s. Namely, using the algorithm of Geffert et. al [4] as our starting point, we show that multiway blockwise in-place merging is possible with $\lceil \log k \rceil \cdot n + O((n/s)^2) + O(s \cdot \log s)$ element comparisons and $3 \cdot n + O(s \cdot \log s)$ moves. For $s = \lceil n^{2/3}/(\log n)^{1/3} \rceil$, this gives an algorithm with $\lceil \log k \rceil \cdot n + O((n \cdot \log n)^{2/3})$ comparisons and $3 \cdot n + O((n \cdot \log n)^{2/3})$ moves, and the number of element moves independent on the number of input sequences. (It is also easy to show that the number of comparisons cannot be improved.)

2 Comparisons in a simple multiway merging

To explain how elements are compared, we first solve a simpler task. Assume that we are given an array A, consisting of k sorted subsequences $A_1, A_2, \ldots, A_k$, that are to be merged into a single sorted sequence. The lengths of these subsequences are $n_1, n_2, \ldots, n_k$ respectively, with $\sum_{i=1}^{k} n_i = n$.

Assume also that, together with the given array A, we are also given an extra array B of the same size n, which will be used as an output zone.

During the computation, the algorithm uses auxiliary index variables $i_1, \ldots, i_k$ and o_c, where i_j, for $j \in \{1, \ldots, k\}$, points to the smallest element of the

[1] Throughout the paper, $\log x$ denotes the binary logarithm of x.

sequence A_j not yet processed. This element will be called the *current input element of the j-th sequence*, or simply the *j-th input element*. The index o_c points to the leftmost empty position in the array B.

Then the straightforward implementation of the merge routine proceeds as follows. We find the smallest element not yet processed, by comparing elements at the positions $i_1, \ldots, i_k$, and move this element to the output zone in B. After that, we update the necessary index variables and repeat the process until all the elements have been merged. Implemented this way, each element will be moved just once and the number of comparisons, per each element, will be $k-1$. This gives us $(k-1)\cdot n$ comparisons and n element moves in total.

The number of comparisons can be reduced by implementing a selection tree of depth $\lceil \log k \rceil$ above the k current input elements. Initially, to build a selection tree, $k-1$ comparisons are required. Then the smallest element, not yet processed, can be moved to the output zone. After this, the element following the smallest element in the same subsequence is inserted in the tree and the selection tree is updated. To do this, only $\lceil \log k \rceil$ comparisons are needed. To avoid element moves, only pointers to elements are stored in the selection tree. (For more details concerning this data structure, see [1–3].) The number of moves remains unchanged, but now we have $k-1$ comparisons for the first element and only $\lceil \log k \rceil$ comparisons per each other element. This gives us a total of $(k-1) + \lceil \log k \rceil \cdot (n-1) \leq \lceil \log k \rceil \cdot n + O(1)$ comparisons.

3 Comparisons in a blockwise merging

This section describes one of the cardinal tricks used in our algorithm. Again, we are given the array A consisting of the sorted subsequences $A_1, \ldots, A_k$, to be merged together. We still have the extra array B, used as an output zone.

However, now the entire array A is divided into blocks of equal size s (the exact value of s will be determined later, so that the number of comparisons and moves is minimized) and, before the merging can start, these blocks are mixed up quite arbitrarily. Because of the permutation of blocks in A, we no longer know the original membership of blocks in the input sequences $A_1, \ldots, A_k$.

Still, the relative order of elements inside individual blocks is preserved. Moreover, we shall also assume that $n_1, \ldots, n_k$, the respective lengths of input sequences, are positive integer multiples of s, and hence, before mixing the blocks up, there was always a block boundary between the last element of A_i and the first element of A_{i+1}, for each $i \in 1, \ldots, k-1$.

Before passing further, we define the following relative order of blocks in the array A. Let X be a block with the leftmost and the rightmost elements denoted by x_{L} and x_{R}, respectively. Such block can be represented in the form $X = \langle x_{\text{L}}, x_{\text{R}} \rangle$. Similarly, let $Y = \langle y_{\text{L}}, y_{\text{R}} \rangle$ be an another block. We say that the block X is smaller than or equal to Y, if $x_{\text{L}} < y_{\text{L}}$, or $x_{\text{L}} = y_{\text{L}}$ and $x_{\text{R}} \leq y_{\text{R}}$. Otherwise, X is greater than Y. In other words, the blocks are ordered according to their leftmost elements and, in the case of equal leftmost elements, the elements at the rightmost positions are used as the second order criterion.

Now the modified merging algorithm proceeds as follows. First, using the above block ordering, find the smallest k blocks in the array A. These blocks will initially become the k *current input blocks*, their leftmost elements becoming the k *current input elements*. The j-th current input block will be denoted by X_j, similarly, the j-th current input element by x_j. The positions of current input elements are kept in index variables $i_1, \ldots, i_k$. Above the k current input elements, we build a selection tree. All blocks that are not input blocks are called *common blocks*.

After that, the merging process can proceed in the same way as described in Section 2. That is, using the selection tree, determine i_j, the position of the smallest input element not yet processed, among the k current input elements, and move this element to the output zone in the array B. Then the element positioned immediately on the right of x_j, within the same block X_j, becomes a new j-th current input element, its index pointer is inserted into the selection tree, and the tree is updated, with $\lceil \log k \rceil$ comparisons. This can be repeated until one of the current input blocks becomes empty.

When this happens, i.e., each time the element x_j, just moved to the output zone, was the last (rightmost) element in the corresponding input block X_j, the block X_j is "discarded" and the smallest (according to our relative block ordering) common block not yet processed will be used as the new j-th current input block. The leftmost element in this block will become the new j-th current input element. Since the blocks are mixed up in the array A, we need to scan sequentially all blocks (actually, all blocks not yet processed only) to determine which one of them is the smallest. This search for a new input block consumes $O((n/s)^2)$ additional comparisons: there are at most n/s blocks and such search is activated only if one of the input blocks has been discarded as empty, i.e., at most n/s times. (For the time being, just assume that we can distinguish discarded blocks from those not yet processed, at no extra cost.)

However, before merging, the blocks have been mixed up quite arbitrarily and hence their origin in the input subsequences $A_1, \ldots, A_k$ cannot be recovered. The proof that the above algorithm behaves correctly, that is, the elements are transported to the output zone in sorted order, will be published in the full version of the paper. (** The proof can also be found in the Appendix for the Program Committee. **)

The number of element moves remains unchanged, but now we use $\lceil \log k \rceil \cdot n + O((n/s)^2)$ comparisons, under assumption that we can distinguish discarded blocks from those not yet processed at no extra cost.

4 In-place merging, simplified case

Now we shall convert the above merging algorithm into a procedure working "almost" in-place. More precisely, we are again given the array A containing the sorted subsequences $A_1, \ldots, A_k$, of respective lengths $n_1, \ldots, n_k$, with $\sum_{i=1}^{k} n_i = n$. All these lengths are positive integer multiples of the given parameter s.

However, we no longer have a separate array B of size n. Instead, we have some extra $k \cdot s$ elements positioned at the very end of the array A, behind A_k. The elements in this small additional area are greater than any of the elements in $A_1, \ldots, A_k$. During the computation, they can be mixed with other elements, but their original contents cannot be destroyed. These elements will be called *buffer elements*. To let the elements ever move, we have also one extra location where we can put a single element aside.

The sorted output should be formed within the same array A, in the locations occupied by the input sequences $A_1, \ldots, A_k$. (As a consequence, the buffer elements should also end up in their original locations.) Therefore, the moves are performed in a different way, based on the idea of internal buffering, used in a two-way in-place merging [4]. Nevertheless, the comparisons are performed in the same way as described in Section 3.

4.1 Initiation

Divide the entire array A into blocks of equal size s. Since the lengths of all input sequences $A_1, \ldots, A_k$ are positive integer multiples of s, there is always a block boundary between the last element of A_i and the first element of A_{i+1}, for each $i \in 1, \ldots, k-1$. Similarly, the buffer elements, positioned in the small additional area at the very end, form the last k blocks.

Initially, the last k blocks will be used as *free blocks*, their starting positions are stored in a *free block stack* of height k. After that, the position of one free block is picked out of the stack and this block is used as a so-called *escape block*. We also maintain a *current escape position* e_c, which is initially the position of the first (leftmost) element in the escape block. We create a hole here by putting the buffer element at this position aside.

Now, find the smallest k blocks[2] in the area occupied by $A_1, \ldots, A_k$, according to the relative block ordering defined in Section 3. This can be done with $O(k^2) \leq O(1)$ comparisons, by the use of some k cursors (index variables) moving along in A, since each of the sequences $A_1, \ldots, A_k$ is sorted. The smallest k blocks will initially become the k *current input blocks* $X_1, \ldots, X_k$. For each $j = 1, \ldots, k$, the first element x_j in the block X_j becomes a *j-th current input element*, and its position is kept in the index variable i_j. Above the k input elements, we build a selection tree of depth $\lceil \log k \rceil$. To do that, $k-1 \leq O(1)$ initial comparisons are needed.

The very first block of the array A becomes an *output block* and a position $o_c = 1$ pointing there becomes a *current output position*. The initial output position may — quite likely — coincide with a position of some current input element. Observe that $e_c \bmod s = o_c \bmod s$, which is an invariant we shall keep in the course of the entire computation. All other blocks are called *common blocks*.

In general, the algorithm maintains current positions of the following special blocks: free blocks, the number of which ranges between 0 and k, their leftmost positions are stored in the free block stack; exactly k input blocks, the current input positions inside the respective blocks are stored in the index variables $i_1, \ldots, i_k$; one output block with the current output position o_c inside this block; and one escape block with the current escape position e_c inside. The values of o_c and e_c are synchronized modulo s.

Usually, the optional free blocks, the k current input blocks, the output block, and the escape block are all disjoint, and the merging proceeds as described in Section 4.2. However, after the initiation, the output block may overlay one of the current input blocks, if the leftmost block in A_1 has been selected as an input block. If this happens, the current output position coincides with a position of one of the current input elements, and the computation starts in a very special mode of Section 4.9.

[2] Picking simply the leftmost blocks in the sequences $A_1, \ldots, A_k$ would do no harm. In addition, this would not require any initial element comparisons. However, we are presenting the algorithm in a form that is suitable for application in the general case, comparing elements in accordance with the strategy presented in Section 3.

4.2 Standard situation

The standard situation is illustrated by Fig. 1. During the computation, the $k \cdot s$ buffer elements can be found at the following locations: to the left of the j-th input element x_j in the j-th input block X_j, for $j \in \{1, \ldots, k\}$, to the right of e_c in the escape block, with the hole at the position e_c, and also in free blocks, consisting of buffer elements only.

The elements merged already, from all the input blocks, form a contiguous output zone at the very beginning of A, ending at position $o_c - 1$. Hence, the next element to be output will go to the position o_c in the output block.

All elements not merged yet are scattered in blocks between the output zone and the end of the array A. The permutation of these blocks is allowed, however, elements to be merged keep their relative positions within each block. On the other hand, the origin of the blocks in the subsequences $A_1, \ldots, A_k$ cannot be recovered. So optional free blocks, input blocks, escape block, and common blocks can reside anywhere between the output zone and the end of the array A.

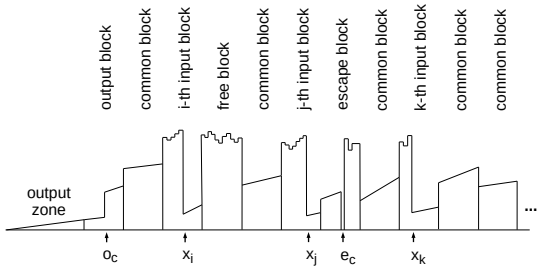

Fig. 1. Standard situation

The output block spans across the current output position o_c, so its left part belongs to the output zone. As the output grows to the right, the elements lying to the right of o_c are moved from the output block to the corresponding positions in the escape block, i.e., to the right of e_c. The positions of o_c and e_c are synchronized, i.e., we have always $o_c \bmod s = e_c \bmod s$. Hence, the relative positions of escaping elements are preserved within the blocks. Moreover, o_c and e_c reach their respective block boundaries at the same time.

Now we are ready for merging. Using the selection tree, we determine x_j, the smallest element among the k current input elements in the blocks $X_1, \ldots, X_k$, and move this element to the output zone as follows:

Step A. The element at the position o_c in the output block escapes to the hole at the position e_c.

Step B. The smallest input element x_j not yet processed is moved from the position i_j to its final position at o_c.

Step C. A new hole is created at the position $e_c + 1$ by moving its buffer element to the place released by the smallest input element just moved. After that, all necessary index variables are incremented and the selection tree is updated.

This gives 3 moves and $\lceil \log k \rceil$ comparisons per each element transported to its final location. Now there are various special cases that should be detected and handled with care. All exceptions are checked up on after the execution of Step B, in the order of their appearance, unless stated otherwise. Most of the exception handling routines replace Step C by a different action.

4.3 Escape block becomes full

If the rightmost element of the output block is moved to the last position of the escape block, the new hole cannot be created at the position $e_c + 1$ in Step C. Instead, one free block at the top of the stack becomes the new escape block and a new hole is created at the beginning of this block. This is accomplished by removing its starting position from the free block stack and assigning it to e_c.

The subsequent move of the buffer element from the new position of e_c to the place released by the smallest input element does not increase the number of moves; it replaces the move in Step C. The selection tree is updated in the standard way.

It should be pointed out that, at this moment, there does exist at least one free block in the stack. Assume, for example, that the j-th input element x_j has just been transported to the output zone. After that, we have $r_j \in \{1, \ldots, s\}$ buffer elements in the j-th input block X_j, including the hole, but $r_h \in \{0, \ldots, s-1\}$ buffer elements in other input blocks X_h, for each $h \in \{1, \ldots, k\}$, $h \neq j$, since each input block, except for X_j, contains at least one input element. Moreover, the escape block is full, and hence it does not contain any buffer elements at all. Assuming there is no free block available, this gives at most $s + (k-1) \cdot (s-1) < k \cdot s$ buffer elements in total. But this is a contradiction, since the number of buffer elements, including the hole, is always equal to $k \cdot s$.

4.4 Current input block becomes empty

We check next whether the smallest element x_j, just moved from the position i_j to the output zone, was the last element of the corresponding input block X_j.

If so, we have an entire block consisting of buffer elements only, with hole at the end after Step B. This hole is filled in the standard way, described in Step C, but the old input block X_j becomes a free block and its starting position is saved in the stack. Since we have $k \cdot s$ buffer elements in total, a stack of height k is sufficient.

Next, we have to find a new j-th input block X_j, and assign a new value to i_j. Since the blocks are mixed up, we scan sequentially the remaining common blocks to determine which common block should become the new j-th current input block. The smallest common block, according to the block ordering introduced in Section 3, is the next one to be processed. As already shown in Section 3, the elements are transported to the output zone in sorted order even though this strategy does not necessarily pick up the j-th input block from the j-th input sequence A_j.

Free blocks, as well as all remaining current input blocks, are ignored in this scanning. Moreover, the elements to the left of e_c in the escape block (if not empty) together with the elements to the right of o_c in the output block are viewed as a single logical block. In a practical implementation, we can start with the leftmost escape-block element and the rightmost output-block element as a starting key and search the rest of the array for a common block with a smaller key.[3] If the logical block composed of the left part of the escape block and the right part of the output block should be processed next, the program control will be switched to the mode described in Section 4.5.

If the escape block is empty, then both e_c and o_c point to the beginning of their respective blocks. Then the escape block is skipped and the output block is handled as a common block, so we may even find out that the new input block should be located at the same position as the output block. This special mode is explained in Section 4.9.

The search for new input blocks costs $O((n/s)^2)$ additional comparisons: there are $O(n/s)$ blocks in total and such search is activated only if one of the input blocks is exhausted, i.e., at most $O(n/s)$ times. The same upper bound holds for arithmetic operations with indexes as well.

[3] It is quite straightforward to detect whether a block beginning at a given position ℓ is common: the value of ℓ must not be saved in the free block stack, and $\lfloor \ell/s \rfloor$ must be different from $\lfloor i_1/s \rfloor, \ldots, \lfloor i_k/s \rfloor$ (excluding $\lfloor i_j/s \rfloor$), and also from $\lfloor e_c/s \rfloor$. For each given block, this can be verified in $O(k) \leq O(1)$ time, performing auxiliary arithmetic operations with indexes only, but no element comparisons or moves.

4.5 One of the input blocks overlays the escape block

If the common block that should be processed next is the logical block composed of the left part of the escape block and the right part of the output block, then both the new current input block X_j and the escape block are located within the same physical block. Here x_j is always positioned to the left of e_c and the buffer elements are both to the left of x_j and to the right of e_c.

Once the position of x_j is properly initiated, all actions are performed in the standard way described in Section 4.2. That is, the elements are transported from the output block to the position of e_c, from the input blocks to the position of o_c, and buffer elements from e_c+1 to locations released in the input blocks. Since e_c moves to the right "faster" than does i_j, this special case returns automatically to the standard mode as soon as e_c reaches a block boundary. Then the escape block separates from the current input block X_j as described in Section 4.3.

4.6 Output block overlays the escape block

Next we check whether the output zone, crossing a block boundary, does not bump into any "special" block. It is easy to see that this may happen only if e_c points to the beginning of the escape block that is empty, using the fact that the positions of o_c and e_c are synchronized and that the special handling of Section 4.3 is performed first.

Now consider that the output block overlays the escape block, i.e., they are both located within the same physical block. In this mode, we always have $o_c = e_c$. The element movement corresponds now to a more efficient scheme:

Step B'. The smallest input element x_j not yet processed is moved to the hole at the position $o_c = e_c$.

Step C'. A new hole is created at the position $o_c+1 = e_c+1$ by moving its buffer element to the place released by x_j. Then all necessary index variables are incremented and the selection tree is updated.

Step A is eliminated, since $o_c = e_c$. This mode is terminated as soon as o_c and e_c reach a block boundary. We also need a slightly modified version of the routine described in Section 4.4. If one of the input blocks becomes empty, it becomes free as usual, but the combined output/escape block is skipped in the search for the next input block.

4.7 Output block overlays a free block

If the output zone crosses a block boundary and the value of o_c is equal to some f_ℓ, the leftmost position of

a block stored in the free block stack, the new output block and the corresponding free block are overlaid. This can be verified in $O(k) \leq O(1)$ time. By the same argument as in Section 4.6, we have that e_c must point to the beginning of an empty escape block.

Therefore, we can easily swap the free block with the escape block by swapping the pointers stored in f_ℓ and e_c, since both these blocks contain buffer elements only. Second, one move suffices to transport the hole from one block to another. Note that this element move is for free, we actually save some moves because the next s transports to the output zone will require only $2s$ moves, instead of $3s$ as in the standard case. Thus, the program control is switched to the mode described in Section 4.6.

4.8 Output block overlays a current input block

If the output position o_c points to some X_j after crossing a block boundary, the output block overlays the j-th input block X_j. Again, by the argument presented in Section 4.6, o_c can point to the beginning of an input block only if e_c points to the beginning of an empty escape block. There are now two cases to consider.

First, if the j-th current input element x_j is the leftmost element of X_j, the program control is switched immediately to the special mode to be described in Section 4.9.

Second, if x_j is not the leftmost element of X_j, we dispose of the empty escape block as free by storing its starting position e_c in the stack, create a hole at o_c by moving a single buffer element from the position o_c to e_c, and overlay the output block by a new escape block, by assigning the value of o_c to e_c. The additional transportation of the hole is for free, not increasing the total number of moves, because we can charge it as (nonexistent) Step A for the next element that will be transported to the output zone. Since x_j is not placed at the beginning of the block, we can guarantee that at least one transport to the output will use only two moves in the next future.

This special mode can be viewed as if *three blocks* were overlaid, namely, the output, escape, and the current input block X_j. The buffer elements are between the hole at $e_c = o_c$ and the current input element x_j. The elements are moved according to Step B' and Step C' of Section 4.6. However, there is a different exception handling here.

(1) If the rightmost input element of this combined block has been transported to the output zone, then the input block X_j separates from the output/escape block, since we search for the next input block to be processed. But here, unlike in Section 4.4, the combined output/escape block is not disposed of as free, moreover, it is skipped out during the search. The program control is switched to the mode of Section 4.6 as the output and escape blocks are still overlaid.

(2) Let us now consider that this combined block becomes full. This may happen only if, for some $h \neq j$, an element x_h from another input block X_h is moved to the output zone and, after Step B', the output position o_c "bumps" into x_j. In this case, we take one free block from the top of the stack and change it into a new escape block. We definitely have at least one free block available, since we disposed one block as free at the very beginning of this mode. The hole, located in X_h at the position of the last element transported to the output, jumps to a position e_c in the new escape block, so that $e_c \bmod s = o_c \bmod s$. This move replaces Step C' for the last element just merged. Hence, it does not increase the total number of moves. Then we follow the instructions of Section 4.9.

4.9 Output zone bumps into a current input element

The program control can jump to this special mode from several different places (Sections 4.1, 4.4, and two different places in Section 4.8). In any case, we have an empty escape block, containing the hole and buffer elements only. The output block and a block X_j, which is one of the input blocks, are overlaid. Moreover, there is no room in between, the output position o_c is pointing to the current input element x_j. The position of hole in the escape block is synchronized with o_c, i.e., we have $e_c \bmod s = o_c \bmod s$.

As long as the elements to be output are selected in the input block X_j, they can be moved to the output zone. This needs no actual transportation, just the positions of o_c and i_j are moved synchronously to the right. To keep e_c synchronized with o_c, we move the hole along the escape block in parallel, which gives us one move per element. There are two ways out of this loop.

(1) If o_c and i_j reach the block boundary, we simply search for the next input block to be processed; the current configuration is the same as if, in the standard mode, o_c, e_c, and i_j reached the block boundaries at the same time (with the old input block X_j disposed of as free, by Section 4.4). Thus, unless something "exceptional" happens, the program control returns to the standard mode. (The possible exceptions are those discussed in Sections 4.6–4.8, and 4.10.) The single move required to place the hole back to the begin-

ning of the escape block is for free, it substitutes Step C for the last element merged.

(2) If the element to be transported to the output zone is an element x_h from another input block X_h, for some $h \neq j$, some rearrangements are necessary. Recall that the hole position e_c in the escape block is synchronized with o_c, i.e., we have $e_c \bmod s = o_c \bmod s$. First, the input element x_j is moved from position o_c to position e_c. Now we can transport x_h to the output position o_c. Finally, a new hole is created[4] at the position e_c+1 by moving its buffer element to the place released by x_h.

The result is that the current input block X_j, overlaid by the output block, jumps and overlays the escape block. Thus, the control is switched to the mode of Section 4.5.

Clearly, this rearrangement needs only three moves. Since one more element has been transported to the output zone, the number of moves is the same as in the standard case.

4.10 Common blocks are exhausted

If one of the current input blocks becomes empty, but there is no common block to become a new input block, the above procedure is stopped. At this point, the output zone, consisting of the elements merged already in their final locations, is followed by a residual zone of size n' starting at the position o_c. This zone consists of the right part of the output block, $k-1$ unmerged input blocks, at most k free blocks, and one escape block. Thus, the total length of this residual zone is $n' \leq s + (k-1)\cdot s + k\cdot s + s = (2k+1)\cdot s$.

The residual zone can be sorted by the use of Heapsort (including also the buffer element put aside at the very beginning of the computation, to create a hole). This will cost only $O(k\cdot s\cdot\log(k\cdot s)) \leq O(s\cdot\log s)$ comparisons and the same number of moves [5–9]. Alternatively, we could also use an algorithm sorting in-place with $O(s\cdot\log s)$ comparisons but only $O(s)$ moves [10].

Now we are done: the buffer elements are greater than any other element, and hence the array now consists of the subsequences $A_1, \ldots, A_k$ merged into a single sorted sequence, followed by a sorted sequence of buffer elements.

[4] Unless the position e_c+1 itself is across the block boundary. If x_j is moved to the rightmost position in the escape block, the escape block jumps immediately and one free block becomes a new escape block. This nested exception thus returns the algorithm to the standard mode; all "special" blocks now reside in pairwise disjoint regions. However, we jump to the point where the standard routine checks the exceptions of Sections 4.4–4.10. Among others, we have to check whether the input block X_h has not become empty, or if the output zone, just crossing a block boundary, has not bumped into any other "special" block again.

4.11 Summary

Summing up the costs paid for maintaining the selection tree, transporting the elements to the output zone, searching for smallest input blocks, and for sorting the residual zone, it is easy to see that the above algorithm uses $\lceil\log k\rceil\cdot n + O((n/s)^2) + O(s\cdot\log s)$ element comparisons and $3\cdot n + O(s\cdot\log s)$ moves. For $s = \lceil n^{2/3}/(\log n)^{1/3}\rceil$, this gives an algorithm with $\lceil\log k\rceil\cdot n + O((n\cdot\log n)^{2/3})$ comparisons and $3\cdot n + O((n\cdot\log n)^{2/3})$ moves.

5 Conclusion

In this paper we have shown that k-way blockwise in-place merging can be accomplished efficiently with almost optimal number of element comparisons and moves. Moreover, the number of element moves is independent on k, the number of input sequences. Note that this algorithm does not merge stably, that is, the relative order of equal elements may not be preserved. Whether there exist a stable multiway blockwise in-place merging algorithm is left as an open problem.

We conjecture that, using the algorithm described here as a subroutine, it is possible to devise an asymptotically efficient multiway in-place merging algorithm. We dare to formulate this conjecture since the work on such algorithm is currently in progress.

References

1. Katajainen, J., Pasanen, T.: In-Place Sorting with Fewer Moves. Inform. Process. Lett. **70** (1999) 31–37
2. Katajainen, J., Pasanen, T., Teuhola, J.: Practical In-Place Mergesort. Nordic J. Comput. **3** (1996) 27–40
3. Katajainen, J., Träff, J. L.: A Meticulous Analysis of Mergesort Programs. Lect. Notes Comput. Sci. **1203** (1997) 217–28
4. Geffert, V., Katajainen, J., Pasanen, T.: Asymptotically Efficient In-Place Merging. Theoret. Comput. Sci. **237** (2000) 159–81
5. Carlsson, S.: A Note on Heapsort. Comput. J. **35** (1992) 410–11
6. Knuth, D. E.: The Art of Computer Programming, Vol. 3: Sorting and Searching. Addison-Wesley, Second edition (1998)
7. Schaffer, R., Sedgewick, R.: The Analysis of Heapsort. J. Algorithms **15** (1993) 76–100
8. Wegener, I.: Bottom-Up-Heapsort, a New Variant of Heapsort Beating, on an Average, Quicksort (If n Is Not Very Small). Theoret. Comput. Sci. **118** (1993) 81–98

9. Williams, J. W. J.: Heapsort (Algorithm 232). Comm. Assoc. Comput. Mach. **7** (1964) 347–48
10. Franceschini, G., Geffert, V.: An In-Place Sorting with $O(n \cdot \log n)$ Comparisons and $O(n)$ Moves. J. Assoc. Comput. Mach. **52** (2005) 515–37

Appendix for the Program Committee

In this appendix, we give a proof of correctness of the blockwise merging algorithm described in Section 3.

Recall that the algorithm starts with the smallest k blocks of the array A and, each time one of the k input blocks becomes empty, the smallest common block not yet processed becomes the new input block, not taking into account its origin in one of the sequences $A_1, \ldots, A_k$.

Let us assume, for contradiction, that an element x_{C} has just been transported to the output zone, but there still exists an unprocessed element y, such that $y < x_{\mathrm{C}}$. Clearly, the element x_{C} was a current input element in some current input block. Originally, this input block was represented in the form $X = \langle x_{\mathrm{L}}, x_{\mathrm{R}} \rangle$, which gave its relative block order. Here x_{L} represents the original leftmost element of X (transported to the output even earlier than x_{C}) and x_{R} the rightmost element (still residing in X). Since all blocks are sorted, we have that $x_{\mathrm{L}} \leq x_{\mathrm{C}} \leq x_{\mathrm{R}}$ (not excluding the possibility that x_{C} coincides with x_{L} or x_{R}). Similarly, the element y resides in a block characterized by $Y = \langle y_{\mathrm{L}}, y_{\mathrm{R}} \rangle$, with $y_{\mathrm{L}} \leq y \leq y_{\mathrm{R}}$. (The current status of the original leftmost element y_{L} is not known: it can still reside in Y but, potentially, it may be a part of the output zone already.) Now there are the following possibilities to consider:

(a) The block Y is one of the current input blocks (not excluding the possibility that Y coincides with X). This case is straightforward. Let y_{C} denotes the current input element in the input block Y. Clearly, $y_{\mathrm{C}} \leq y$, since y has not been processed yet. But then $y_{\mathrm{C}} \leq y < x_{\mathrm{C}}$, that is, $y_{\mathrm{C}} < x_{\mathrm{C}}$, which is a contradiction, since x_{C} has just been moved to the output, and hence determined to be the smallest one among all current input elements.

(b) The block Y is one of the common blocks (not yet being processed) and, at the present moment, all k current input blocks have their origin in different input sequences $A_1, \ldots, A_k$. This does not necessarily mean that the i-th input block X_i originated from A_i, since it could have its origin in a different sequence. Nevertheless, this does imply that one current input block $Z \in \{X_1, \ldots, X_k\}$ originated from the same sequence A_ℓ as did the block Y.

Let $Z = \langle z_{\mathrm{L}}, z_{\mathrm{R}} \rangle$ represents the original characterization of this current input block, and let z_{C} be its current input element. Clearly, $z_{\mathrm{L}} \leq z_{\mathrm{C}} \leq z_{\mathrm{R}}$. Using the fact that the block Z has been selected as an input block prior to selecting the block $Y = \langle y_{\mathrm{L}}, y_{\mathrm{R}} \rangle$, and that these two blocks originated from the same sorted sequence A_ℓ, we have that $z_{\mathrm{R}} \leq y_{\mathrm{L}}$. (For, if $z_{\mathrm{R}} > y_{\mathrm{L}}$, the sequence A_ℓ could not be sorted.) But this gives

that $z_C \leq z_R \leq y_L \leq y < x_C$. Therefore, $z_C < x_C$, which is a contradiction, since the element x_C has been determined to be the smallest one among all current input elements.

(c) Finally, let Y be one of the common blocks and, at the present moment, at least two current input blocks have their origin in the same input sequence. That is, we have $V, W \in \{X_1, \ldots, X_k\}$, coming from A_ℓ, for some $\ell \in \{1, \ldots, k\}$. Let the respective characterizations of these two blocks be $V = \langle v_L, v_R \rangle$ and $W = \langle w_L, w_R \rangle$, their respective current input elements be v_C and w_C. Clearly, $v_L \leq v_C \leq v_R$ and $w_L \leq w_C \leq w_R$. Without loss of generality, we can also assume that the block V was smaller than or equal to W. But then $v_R \leq w_L$, since these two blocks came from the same sorted sequence A_ℓ. Moreover, at the present moment, the element x_C has just been selected as the smallest one from among all current input elements, and hence we also have that $x_C \leq v_C$. Putting these facts together, we get $y_L \leq y < x_C \leq v_C \leq v_R \leq w_L$. Therefore, $y_L < w_L$, which contradicts the fact that the block W had been selected as an input block prior to selecting the block Y.

Summing up, all cases have led us to contradictions, and hence the elements are always transported to the output zone in sorted order.[5]

[5] Notice that the algorithm would behave correctly even if the number of used current input blocks exceeded k, the number of original input sequences. This only eliminates Case (b) in the proof of correctness.