In-Place Merging Algorithms

Denham Coates-Evelyn
Department of Computer Science
Kings College, The Strand
London WC2R 2LS, U.K.
denham@dcs.kcl.ac.uk

January 2004

Abstract

In this report we consider the problem of merging two sorted lists of m and n keys each in-place. We survey known techniques for this problem, focussing on correctness and the attributes of Stability and Practicality. We demonstrate a class of unstable in-place merge algorithms that uses block rearrangement and internal buffering that actually does not merge in the presence of sufficient duplicate keys of a given value. We show four relatively simple block sorting techniques that can be used to correct these algorithms. In addition, we show relatively simple and robust techniques that does stable local block merge followed by stable block sort to create a merge. Our internal merge is base on Kronrod's method of internal buffering and block partitioning. Using block size of $O(\sqrt{m+n})$ we achieve complexity of no more than $1.5(m+n)+O(\sqrt{m+n}\lg(m+n))$ comparisons and $4(m+n)+O(\sqrt{m+n}\lg(m+n))$ data moves. Using block size of $O((m+n)/\lg(m+n))$ gives complexity of no more than $m+n+o(m+n)$ comparisons and $5(m+n)+o(m+n)$ moves. Actual experimental results indicates $4.5(m+n)+o(m+n)$ moves. Our algorithm is stable except for the case of buffer permutation during the merge, its implementation is much less complex than the unstable algorithm given in [26] which does optimum comparisons and $3(m+n)+o(m+n)$ moves. Buffer extraction and replacement is the only additional cost needed to make our algorithm stable.

Key Terms: Internal Merge, Stable Merge, Block Rearrangement, Internal Buffer,

INTRODUCTION

1 The Ranking Problem

A set of data values can be ranked in space by their relative size by
(1) A process of pair-wise comparison followed by data move operation(s) or
(2) A logical or mathematical method whereby an individual data value location is determined and then followed by data move operations.
From this it can be easily seen that the process of ranking a set of data values requires data space and processing time that are related in some form to the method used to achieve this objective. Ranking data values by the first method is commonly termed as Comparison base sorting. The second method is called Hashing. The first method is a common solution to the ranking problem as it allows minimal restriction on the type and range of data values to be ranked. We seek a solution to the ranking problem that can be implemented on a digital computer system using minimum data space and processor time.

It is well known that there is a minimum limit to the number of comparisons that can be done in order to rank a set of N data values using the comparison method. This minimum number of comparisons is known as the theoretical lower limit and is easily shown to be $N \log_2(N) - (N-1)$. The implication of this is that you can not design an algorithm that will do less than this number of comparisons on average to rank a set of N data values. Note that the theoretical lower limit does not place a limit on the amount of memory space available for ranking. Therefore, it has remain a big challenge in Computer Science to develop an algorithm that will rank N data values using the lower limit on the number of comparisons and a constant amount of extra memory space. Research and development in Computer Science to solve this problem have been going on for over 50 years. Best known algorithms that approach the lower limit of $N \log_2(N) - (N-1)$ on the number of comparisons using constant amount of extra memory and no more than a constant amount of $N \log_2(N)$ data moves are listed in the attached bibliography. However these algorithms does not achieve the exact lower limit and in some instance have other undesirable attributes as explained below. From now on we use the term sort to mean a process whereby a set of data values are ranked by the method of pair-wise comparisons of data values followed by data move operation(s). From this, we see that the desirable characteristics of a good sorting algorithm are

(1) The number of comparisons and data moves done to sort N data values is about a constant amount of $N\log_2(N)$. In this case the algorithm is said to be *optimum* and is termed as an $O(N\log_2(N))$ algorithm.

(2) The extra memory space required by the algorithm apart from that required for storing the N data values is the same regardless of the value of N. In this case, the algorithm is said to be *in-place*.

1.1 The Big Question

What is the most efficient general algorithm to order a set of data values in the memory of a digital computer base on their relative size? Is there a *stable* in-place algorithm that achieves the theoretical lower bound on the number of comparisons and data move operations?
Some Answers
Use a general algorithm that has the following attributes
(1) Does ranking by comparison
(2) Not difficult to programme

In addition, we would like our sort algorithm to have the following attributes.
(a) Number of comparison done is proportional to $N\log_2(N)$. With small proportionality constant.
(b) Uses the same amount of extra memory regardless of the value of N.
(c) Keep equal set of data values in the same order at the end of sorting.

More recent research work in this area have highlighted the attribute of *stability*, its implication on the correctness and complexity of other data processing activities and its implication on the correctness of particular sorting method. A sorting method is said to be stable if it leaves a set of equal data values in the same order as they were before at the end of sorting. We use an example to illustrate this important attribute later on. we give more concise definition of italised terms later on. As we see further on, stability has proven to be a critical attribute for the correctness of in-place *merge* algorithms.

1.2 A General Statement About Merging as it Relates to Sorting

Merging is the process whereby two pre-sorted lists of m and n data values each are combine in a systematic manner to create a single sorted output list of $m + n$ data values. This sorted list consists of the original values in the two presorted lists. The most natural technique for merging two lists of sorted data values is to compare the smallest values in both lists and then output the smaller of the two values to a new list. This process is repeated for the remaining values in both lists until one of the list become exhausted. The remaining values in the non-exhausted list are appended to the output list in sorted order. Merging in this way requires $m + n - 1$ comparisons and $m + n$ data moves to create a sorted list of $m + n$ data values. In addition, this lower limit is guaranteed regardless of the nature of the input permutation. We refer to this merge technique as a natural merge operation. In some place, sorting is described as a process of information gathering followed by data movement. Sorting by merging clearly demonstrate this notion. Merging have the favorable attributes of using no more than $m+n-1$ comparisons and $m+n$ data moves to merge two pre-sorted lists of m and n keys each. Therefore, it is possible to use merge in an iterative or recursive manner to sort a list of N data values in a stable manner using no more than $N \log_2 N - (N - 1)$ comparisons and no more than about the same number of data moves. We use the term Merge-sort to describe a sort implemented in this manner. The reader should take note of the following assumption on the above explanation. I.e. each data value in the input lists to be merged is allocated a fixed amount of memory. Therefore, the issues pertaining to the merge of variable length records does not apply. This is dealt with in a further discussion.

Unfortunately, the merge operation as described above require the use of $O(m + n)$ extra memory as output for the newly created merged list. We note that each data item or record that is moved during the sort (or merge) is allocated a fixed amount of memory locations. Therefore, the time used for moving each data item is fixed, a data ietm is easily moved to a new location and the location of any two data items are easily interchange during a sort (or merge) operation where data interchnage is required. However, the merge operation as decribed above does not require the interchange of data items. Therefore sorted lists consisting of variable lenghth data items can be merged using this method. There are no known internal merge technique where this is the case.

John Von Neuman first proposed merging in 1945 as a method for sorting data on digital computer systems that use the stored program concept. At this time most existing merge techniques used less expensive external memory as their output medium during the merge operation. Merge-sort is used for sorting large data files on systems where internal Random Access Memory (RAM) is more expensive than external magnetic mediums. The time required for input and output operations during the merge far exceed that required for internal comparisons and data moves. This creates a time bottleneck during the merge operation. Hence, the main disadvantage of using external memory is that input and output operations during the merge significantly reduce the overall speed. Here we refer to an external merge as a technique that uses external memory for output. Many external merge techniques attempt to increase the efficient use of processor time and memory resource. Many of these techniques attempt to syncronise internal data processing activities with input and output operations. For all practical purpose, this increases the complex requirement of the operating systems and the physical configuration of hardware systems on which such techniques are implemented.

A significant theoretical breakthrough took place in the late 1960s'when M.A. Kronrod [16] showed that two presorted lists of m and n keys each can be merge internally using constant amount of extra memory and $O(m+n)$ comparisons and data moves. M. A. Kronrod presented the key concepts of the *Internal Buffer* and *Block Rearrangement*. Since then, many new techniques have appeared in various related academic publications. The original emphasis on new techniques has been their theoretical complexity. However, many of these techniques provide useful insight on new methods for devising better algorithms of a more practical nature.

Although most of these early techniques are optimum on the number of comparisons and data moves, they tend to have large proportionality constants that renders them impractical. This results mainly from large amount of data moves. In addition, some of these algorithms are not stable and in some instance will not perform a proper merge. We illustrate instances for selected algorithms that actually do not produce a merge. We highlight the importance of stability during block sorting to the merge actually taking place.

In addition to doing no more than $m+n-1$ comparisons and $m+n$ data moves when merging two presorted lists of m and n data values each, natural merge is also stable. However, the basic ideas presented by M.A. Kronrod results in unstable merge and increase comparisons and data moves. More recent techniques attempt to achieve stability and to reduce the amount of data moves during the merge. Hovarth [9] developed a stable optimum in-place merge algorithm that uses the key concepts in Kronrod's algorithm. However, the algorithm does data modification and has a relatively large proportionality constant. In order to achieve stability, Hovarth uses $O(\log_2(m+n))$ data bits as pointers to encode the correct movement of data values during the merge. Later on Trab Pardo derived a stable optimum algorithm that does not use key modification [22]. It is clearly demonstrated that stable block sorting followed by stable local block merge does not necessarily lead to a stable merge. However, as we clearly show in the following discussion, stable block sorting is a sufficient condition that guarantees correctness of these algorithms.

A modified version of Kronrod's algorithm is given by Huang & Langston [10]. Their algorithm is unstable and does no more than $1.5(m+n)+O(\sqrt{m+n}\log_2(m+n))$ comparisons and $6(m+n)+O(\sqrt{m+n}\log_2(m+n))$ data moves to merge two presorted lists of m and n data values each. The average number of data moves is reduced by moving the buffer across series of blocks that are sorted by their last element in non-decreasing order. The need to do a block swap at the start of each local merge is eliminated by locating natural runs in the block sorted sequence and then merging with a second series of size no more than k. Where k is the number of elements in each block. A merge algorithm that achieves the lower bound on the number of comparisons with $m<n$ is given in [24]. This algorithm uses the main ideas in Mannila's and

Ukkonen's technique [18] to achieve the lower bound.

Consequent to this, V. Geffert et al [26] have given theoretical results proving that optimum in-place merge can be done using $3(m+n)+o(n)$ data moves and $m(t+1)+n/2^t+o(m)$ comparisons. Where $m \leq n$ and $t = \lfloor \log_2(n/m) \rfloor$. An implementation algorithm is given. However, this algorithm is relatively complicated. In addition, the upper bound on the number of data moves increases to $5n + 12m + o(m)$ when the algorithm is made stable. We present two relatively simple and robust algorithms for merging two presorted lists of m and n data values each that does no more than $(m+n)+o(m+n)$ comparisons and $5(m+n)+o(m+n)$ data moves for block size of $O(\frac{m+n}{\log_2(m+n)})$ and no more than $1.5(m+n)+o(m+n)$ comparisons and $4(m+n)+o(m+n)$ data moves using block size of $O(\sqrt{m+n})$. Our algorithms are stable except for the case of buffer permutation during the local block merge operation. Making the algorithm stable is a relatively simple task that can be implemented using any well known buffer extraction technique.

The remaining section of the paper is organized as follows: In Section 2 we define our main terminology, notations and convention. We also give a general overview of well-known results and techniques used in this discussion. Section 3 is a brief overview of Kronrod's and Huang & Langston's algorithms along with a general proof of their correctness for the case where the number of duplicates for any given value is less than the block size. We also do computational complexity analysis and tabulate experimental results from their implementation. The number of comparisons and data moves is used as the basis of our metrics. In section 4 we present four optimum and stable block sorting techniques. We show that stable block sorting with minimum data moves is relatively trivial. In section 5 we present our merge algorithms. This includes complexity analysis on the number of comparisons and data moves. We also give tabulated results from its implementation. Section 6 summarizes our main results with concluding remarks and suggestions for further development.

2 Terminology and Definitions

When a set of data values are sorted, they are ranked in order of their relative size. For example, the following sequence of integer values is said to be sorted in non-decreasing order $-10, -8, -8, 1, 3, 7, 12, 12$. If we were to place the values in reverse order by putting the largest value in the lowest position followed by the next largest value and so on, then we say that the values are sorted in decreasing order. Placing a set of data values in sorted order in the memory of a digital computer is important to other internal operations such as searching, data insertion, data extraction, data encryption, data compaction and so on.

It has been shown that the lower limit on the number of comparison and data move operations when merging a set of m and n pre-sorted data values is $m + n - 1$ and $m + n$ respectively. However, the lower limit on the number of comparisons is further reduce to $m^t + n/2t$, where $t \approx \log_2(m/n)$ and $m < n$. This is done by taking advantage of the fact that a fast search can be done on the list consisting of the n keys with selected values from the list consisting of m keys by use of the binary search technique. Series of values less than the value used to do the search are then sent as output to the merged list without the need for further comparisons. In this paper we use $\lg(N)$ to mean $\log_2(N)$.

A sort or merge algorithm is said to be *in-place* whenever it does sorting or merging with the use of constant extra memory. In this case, the amount of extra memory required to implement the algorithm does not depend on the number of records in the input list(s).

In addition, the sort (or merge) algorithm is *stable* if it keeps the indices of a sequence of equal values in the input list (in any of the input list) in sorted order at the end of sorting (or merging). Otherwise, the algorithm is said to be unstable. For example, if we have the set of indexed values in the two presorted lists A and B as follows.

A	1_1	2_1	2_2	2_3	5_1	5_2	7
B	1_2	1_3	2_4	2_5	6	8	9

Output when the merge operation is stable is as follows.

C	1_1	1_2	1_3	2_1	2_2	2_3	2_4	2_5	5_1	5_2	6	7	8	9

Note that, the merge operation that produces the output list C selects values from A in preference to those from B whenever a tie occurs. We define stability in the general sense to mean that, whenever duplicates of a given value occurs both in the A and B lists, the duplicates in A are given smaller indices to those in B and that the indices of equal set of values are always kept in sorted order at the end of the sort or merge operation. Note that in the above example, each duplicate is assigned an index base on its relative location to the first element in the array.

We note that each data item or record that is moved during the sort (or merge) is allocated a fixed amount of memory locations. Therefore, the time used for moving each data item is fixed, a data value is easily moved to a new location and the location of any two data items are easily interchange during a sort (or merge) operation where data interchnage is required. However, the natural merge operation as decribed above does not require the interchange of data items. Therefore sorted lists consisting of variable lenghth data items can be merged using this method. There are no known internal merge technique where this is the case.

During a natural merge operation, each output value is place in its final location by a single *data move*. We define a data move as the process whereby a data value is moved from a selected location in Random Acces Memory (RAM) to a new location. A data move operation on most computer systems requires a fixed number of memory reference and Central Processing Unit (CPU) register manipulation operations depending on the length of each data item and the length of each register in the CPU. We see that natural merge does not require the use of the *swap* operator. We use the following macro to define a swap operation between the two data variables s and t. A variable is defined as a selected sequence of contiguous memory locations that will hold 2^p different data values where p is the number of bits that constitute these memory locations. We see that generally a swap operation uses three data move operations as illustrated by the following macro. In most case, this uses six memory reference operations. However, a swap can be implemented in two moves or equivalently four memory reference operations where the architecture of the CPU allow this.

```
Macro: Swap(s, t)
  HoldVariable = s;
  s = t;
  t = HoldVariable;
End Macro
```

Note that in some programming langugages such as C/C++ we can explicitly reduce the number of moves external to the CPU to 2 by setting HoldVariable as a register variable.

For any indexed sequence T, we denote the number of elements in T as $|T|$ and the value at position j in T as $T[j]$. Where j is an integer that satisfy $0 \leq j \leq |T| - 1$.
We also denote the set of values in T at positions $i, i+1, \cdots j$ inclusively as $T[i : j]$, where $0 \leq i < j \leq |T| - 1$. We define a block in T as a sequence of predefined number of contiguous locations. We also denote the rth block in T as T_r and the value at position i in T_r as $T_r[i]$, where $1 \leq r \leq \lfloor\sqrt{|T|/2}\rfloor$ and $0 \leq i \leq |T_r| - 1$.

Let $T[t] = T\prime[s]$, where $T\prime$ is the sorted sequence derived from the permutation of T and $0 \leq s \leq |T| - 1, 0 \leq t \leq |T| - 1$; We say
1. $T[t]$ is above its final sorted location whenever $t > s$ and
2. $T[t]$ is below its final sorted location whenever $t < s$.
We also denote the upper and lower halves of T as
$T_r = T[0 : r - 1]$ and $T_{2r} = T[r : 2r - 1]$ respectively, where $2r = |T|$.
We say that a sequence of s blocks $T_1, T_2, \cdots T_s$ forms a series whenever $T_j[k-1] < T_{j+1}[k-1]$ and $T_j[k-1] < T_{j+1}[0]$, where $1 < j \leq \lfloor|T|/k\rfloor$ and $k = |T_j|$. Therefore, we see that the values that form a series are in sorted order. Hence, they form a natural run in the merge series.

The merging problem consist of the two presorted sequences A and B, where $|A| = m, |B| = n$ and $m + n = |L|$. The sequences A and B constitute the list L as follows: $A = L[0 : m - 1]$ and $B = L[m : m + n - 1]$. We use an internal buffer consisting of a selection of k locations in

L. Assume that $k < m$ always. At the end of merging A and B, the values in L are sorted in non-decreasing order.

2.1 The Internal Buffer And Block Rearrangement

Generally, the internal buffer consist of a fixed number of contiguous data elements within L. They are used as temporary output locations during the *local block merge* operation. A Local Block merge is a merge of two or more blocks in L. The output from a local block merge goes into the buffer. During the merge operation, consecutive sequences of blocks are selected and then merge into the buffer. Hence, buffer elements are swapped into new locations during a local block merge. As a result of this, the values that constitute the buffer are permuted at the end of a local block merge. If the local block merge operation is stable. Then to maintain overall stability of the merge operation, it is necessary to ensure that all data values in the buffer are unique or that the original permutation of the buffer can be preserve or recovered at the end of local block merging. In Kronrod's original algorithm, the position of the buffer is fixed throughout the merge operation. However, more recent techniques such as [10] allow the buffer to change its location during the Local Block merge operation. In addition, local block merge is done at the end of block rearrangement. We present a merge technique in which local block merge is done before block rearrangement. In addition, the position of the buffer at any instant in time is not localized and the buffer may also split into two non-contiguous sets of locations. Block selection is done concurrently with Local block merge in a manner similar to the natural merge operation described above. This ensures that stability does not affect the correctness of the local block merge operation. Pointers are used to track the location of the buffer during the local block merge.

2.2 Block Size

A regular or full block in A, B or L consist of k fixed elements at a contiguous sequence of memory locations. The value for k is chosen such that the proportionality constant on the number of moves and compare operations remains asymptotically optimal with a practical value. Hence k is chosen as $O(\sqrt{m+n})$ when the number of comparisons or data moves done by the algorithm chosen to implement the block sorting phase of the merge is $O((m+n)^2)$. To do stable block sorting whilst at the same time reduce the number of data moves, we implement block sorting by encoding permutation cycles that determine the final locations of merged blocks. We encode these permutation cycles in either of the following two ways.

(1) With the use of $O(\lg(n))$ extra bits or

(2) Permutation of individual block elements at the end of local block merge to encode the correct sorted location of individual blocks.

Either of these two methods allow us to choose block size of $O((m+n)/\lg(m+n))$ or $O(\sqrt{m+n})$ respectively. We use the variable ν to represent the number of k size blocks in L. Therefore $m+n = \nu \times k + mod(m+n, k)$ or $m+n = \nu \times k + mod(m,k) + mod(n,k)$.

2.3 Block Transformation

Consider the contiguous sequence of blocks ABC represented as

ABC	$A_1, A_2 \cdots A_a$	$B_1, B_2 \cdots B_b$	$C_1, C_2 \cdots C_c$

Where a, b and c are the number of elements in the sequences A, B and C respectively. The permutation CBA is created from ABC by application of a block transformation algorithm. This algorithm can be implemented by a sequence of block reverse operation as illustrated below. This algorithm is used to move contiguous sequences of data values in a stable manner.

2.4 Block Reverse

We define a block reverse operation on $V = \{V_1, V_2, \cdots V_v\}$ as the following transformation on V implemented by the macro BLOCKREVERSE( ) which is defined below. $V\prime = \{V_v, V_{v-1}, \cdots V_2, V_1\}$. Where $v = |V|$.

```
Macro BLOCKREVERSE(i, r)
  INITIALISE s = i; e = r;
       DO:
         Swap(L[s], L[e]);
```

```
            s = s + 1; e = e - 1;
            REPEAT (r - i)/2 times;
        END DO
END Macro
```

A block reverse does a total of $\lfloor v/2 \rfloor$ data swaps. Two adjacent blocks in $L, U = L[i : r]$ and $V = L[r + 1 : j]$ where $0 \leq i < r < j \leq m + n - 1$ are transformed by three series of block reverse operations to create the transformation $UV \longrightarrow VU$.
This transformation uses a total of $u + v - \Delta$ swaps. Where $\Delta = 0$ if both u and v are even, 1 if either but not both u or v is odd, 2 if both u and v are odd, where $u = |U|$.

We seek time and space efficient algorithm(s) to implement block permutation. A recursive version of a block transformation algorithm is given in [19]. This algorithm always do no more than $a + b + c - gcd(a + b, c + b)$ data moves in effecting a block transformation. Where $gcd(a + b, c + b)$ is the greatest common divisor of $a + b$ and $c + b$. To ensure that our merge algorithm is always in-place we use a more recent iterative algorithm given in [3] since it does the same operation in-place.

2.5 Block Sorting

The full blocks in A and B are rearranged so as to create the invariant property given in the next section. Blocks of values are selected into their sorted locations by using a selection sort. Blocks are selected by their mark. The value at a fixed location in each block is chosen as the mark. This is usually the first or the last value in each block. During block sorting, the block that has the smallest mark is chosen as the first block and the block with the next smallest mark is chosen as the next block and so forth. Figure 1 illustrates a possible arrangement of blocks in L at the end of an unstable block sort.

A_1	A_2	A_3	A_4	A_5	A_6	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	B_9

At the end of block sorting we now have the following.

B_1	A_1	B_3	B_2	A_2	B_4	A_3	A_5	B_5	A_4	B_7	B_6	A_6	B_8	B_9

Figure 1
A possible rearrangement of blocks in L at the end of an unstable block sorting.

2.6 Stability And Correctness

The error in Kronrod's method pointed out in [23] highlight the importance of stability for correctness in the presence of duplicates.

As with Kronrod's algorithm, Huang & Langston algorithm will not merge under similar circumstance. For example, if at the end of the block sorting phase we have the following series of values $(888), (458), (129)$, the algorithm will not correctly merge these values during the local block merge. We use single digit whole numbers to represent each value in a block. Open and close brackets delimit blocks of values. Huang & Langston went on to devise a stable version of their algorithm [11]. However, this results in a more complex algorithm with significant increase in the proportionality constant.

Mannila's & Ukkonen's algorithm is easily modified so that it achieves the lower bound on the number of comparisons for merging with $|A| = m < |B| = n$ [18]. The basic idea of the algorithm is a divide and conquer technique originally devised for parallel systems [25]. Basically, the algorithm merge by dividing the A list into $\nu \approx \sqrt{m}$ blocks. The B list is also divided into ν blocks, where the $jthB$ block is located by using the last value in the $jthA$ block to do a binary search to locate a value in the B list that is greater than this value but whose previous adjacent value is less than this value. Where $0 \leq j \leq \nu - 1$. The $jthA$ and B blocks are then moved to the locations they will finally occupy at the end of merging. The two blocks are merge using an internal buffer of size $\approx \nu$. A summary explanation of the algorithm is given in [24, 18]. Unfortunately, the algorithm is unstable and actually will not perform a proper merge under similar circumstance to that given above. For example, if the two A blocks (5555) and (5568) are selected and then merge with the sequence of B values (01234599), a possible outcome from the pair of selection operation could be as illustrated below. Note that the second B block is empty in this case and the remaining values in the B list are 99. Merge blocks $\{(5568), (012345)\}$ and then $\{(5555), ()...(99)\}$. The values are not sorted at the end of the merge operation. The same problem occurs if we select A blocks by their last element as we could have the two sequences of A blocks (1245) and (5555). We show in section 3.2 how to correct the problem of correct block selection in these algorithms in the presence of duplicates of a given value in excess of the block length.

3 Analysis of Kronrod's And Related Algorithms

Our analysis includes a brief overview of Kronrod's and Huang & Langston's algorithms. For Kronrod's algorithm, we use the implementation given in [23]. We determine the upper and lower bounds on the number of comparisons and data moves for both algorithms. We show experimental results on random integer data that confirm these complexity analysis.

3.1 Kronrod's Algorithm

Let $m = n$ and $\nu =$ Number of full blocks in A and B, where $k = \lceil \sqrt{n} \rceil$ and $\nu = \lfloor n/k \rfloor$. Therefore total number of full blocks in L is 2ν. Let $s = mod(n, k)$ the number of element less than k in each undersize blocks.

At the end of local block merging, we use selection sort to sort the k values in the buffer. This uses a total of $k(k-1)/2$ comparisons and no more than k data swaps. The analysis of the algorithm's implementation consist of 7 steps. An internal buffer consisting of the $1st$ full block location in A is used. Totals on the maximum and minimum number of data comparison and data move operations are listed in table 1 below.

Table 1. summary analysis of Kronrods technique

	COMPARISONS		DATA SWAPS	
STEP	Minimum	Maximum	Minimum	Maximum
1	0	0	n	n
2	$\frac{1}{2}\nu(\nu-1)$	$\frac{1}{2}\nu(\nu-1)$	0	$2\nu k$
3	$k(k-1)/2$	$k(k-1)/2$	0	$2k$
4	$(2\nu-2)(2k-1)$	$(2\nu-2)(2k-1)$	$3k(2\nu-2)$	$(2\nu-2)3k$
5	$2s-1$	$2s-1$	$3s$	$3s-1$
6	$2s-1$	$k+s-1$	$3s$	$k+2s$
7	$k(k-1)/2$	$k(k-1)/2$	0	k
	$\approx 5\frac{1}{2}n - 0(\sqrt{n})$	$\approx 5\frac{1}{2}n - 0(\sqrt{n})$	$= 7n - 6k + 6s$	$= 9n - 2k + 5s - 1$

The algorithm was implemented for the case where $n = m$ in ANSI C/C++ code and then compiled and run on a SUN SPARC system in the department of Computer Science at Kings College, University of London. Integer values were generated using standard random number generator functions available in ANSI C++. Our experimental results are consistent with our theoretical analysis as can be seen by comparing the above tabulations with figure 2 below. Experimental test results are tabulated below.

$2n$	n	$Compares(C)$	$Moves(M)$	C/n	M/n
50	25	129	556	5.16	22.24
100	50	230	1230	4.61	24.60
200	100	527	2329	5.27	23.29
500	250	1307	5716	5.23	22.86
1000	500	2253	12107	4.51	24.21
2000	1000	5271	23936	5.27	23.94
5000	2500	13103	60070	5.24	24.03
10000	5000	26893	120023	5.38	24.00
			Averages	5.08	23.65

Figure 2
Average number of comparisons and data swaps
in a merge implemented by Kronrods original technique.

3.2 Huang & Langston's Technique

Table 2 below sumerises our analysis of the algorithm. The algorithm was implemented with slight modification to the buffer extraction operation. This results in $o(s)$ reduction in the number of data comparisons and data moves during buffer extraction.

Table 2 Summary analysis of Huang and Langstons technique

	COMPARISONS		DATA SWAPS	
STEP	Minimum	Maximum	Minimum	Maximum
1	k	k	0	k
2	$k+s-1$	$k+s-1$	0	$k+s$
3	$2s-1$	$2s-1$	$2s$	$3s$
4	0	0	k	k
5	$\nu(\nu-1)$	$\nu(\nu-1)$	0	$(2\nu-1)k$
6	$(2\nu-1)k$	$(2\nu-2)(k+1)$	$(2\nu-1)k$	$(2\nu-1)k$
7	$k(k-1)/2$	$k(k-1)/2$	0	$k(k+1)/2$
	$\approx 3n+3s-\nu-2$ $+O(\sqrt{n}\lg(n))$	$\approx 3n+3s+\nu-4$ $+O(\sqrt{n}\lg(n))$	$=2n+2s$	$=4n+4s+k+1$ $+O(\sqrt{n}\lg(n)))$

The algorithm was coded in ANSI C/C++ and then compiled and tested on a SUN SPARC system in the Computer Science Department at Kings College, University of London. Random integer values were generated using standard C/C++ random number generator functions. Figure 3 tabulates experimental results from the implementation of Huang & Langston's algorithm as described above.

$2n$	n	$Compares(C)$	$Moves(M)$	C/n	M/n
50	25	89	291	3.57	11.64
100	50	168	607	3.35	12.14
200	100	337	1164	3.37	11.64
500	250	820	2905	3.28	11.62
1000	500	1621	6032	3.24	12.06
2000	1000	3288	11942	3.29	11.94
5000	2500	8158	30259	3.26	12.10
10000	5000	16246	60728	3.25	12.15
			Averages	3.33	11.91

Figure 3
Test runs on Huang & Langston's merge algorithm for $m = n$

3.3 Correctness Analysis

We now prove the correctness of Kronrod's algorithm in the presence of unique values. We also do analysis on the nature of distribution of inversions before and at the end of the block sorting.

We define an inversion in L as the ordered pair (i, j) for which $i < j$ but $L[i] > L[j]$, $0 \leq i \leq m - 1 < j \leq m + n - 1$. Let the sequence S be the permutation of L at the end of a block sort and let the sequence $S\prime$ be sorted L. We state the following facts.

Fact 1 *For any pre-sorted sequences A and B, the maximum number of inversions that results from an out of sort element in the sublist B is m.*

Fact 2 *The maximum number of inversions in L is $m(n - 1)$.*

Fact 3 *Any two pre-sorted sequences S and T for which $|S| < |T|$ can be merged using an internal buffer of length $\geq |S|$*

An invariant of the permutation of L at the end of block sorting is that for any given element at position y in a block in L, the value $L[y]$ is no more than $k-1$ positions above its final sorted location. To begin the proof, we first assume the following

(i) $|A| = |B| = n$,

(ii) $|L| = N$ and

(iii) $mod(N, k) = 0$.

We derive the invariant property as follows. We consider the position of a selected value in S and in $S\prime$. The results from our analysis apply to any other value in L that takes part in the block sorting operation. We look at the case where the value in consideration comes from the A list. The case where the value in consideration comes from the B list is given in brackets beside references to the A list. See figure 4.

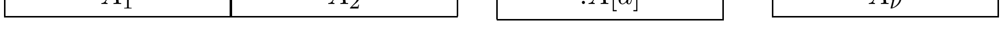
If for some integers x, y, a and b we have $S\prime[x] = S[y] = B[b]$ (or $S\prime[x] = S[y] = A[a]$) and $S\prime[x-1] = A[a]$ (or $S\prime[x-1] = B[b]$), where $0 \leq y \leq x < N-1$ and $0 \leq b \leq a < n-1$. We have the following conditions satisfied on A, B, S and $S\prime$.

i. The number of blocks from A (or B) below the pth block in S is $\leq \lceil a/k \rceil (\lceil b/k \rceil)$. Recall that $S[y]$ is located in the pth block of S. Number of elements from A (or B) to the left of $S[y]$ is $\leq \lceil a/k \rceil k$ (or $\leq \lceil b/k \rceil .k$)

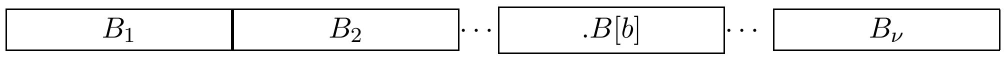
ii. The number of B (or A) blocks in S below the pth block in S is $\lfloor b/k \rfloor$(or $\lfloor a/k \rceil$) number of elements from B (or A) in S below $S[y]$ is b (or a).

From i. and ii. we see that the total number of elements to the left of $S[y]$ is
$\leq b + \lceil a/k \rceil \times k \leq b + (a/k - 1/k + 1) \times k = b + a + k - 1$ or
$\leq a + \lceil b/k \rceil \times k \leq a + (b/k - 1/k + 1) \times k = a + b + k - 1$
$y \leq x + k - 1$. This ends the proof of the invariant property.

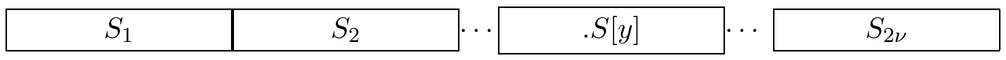
A blocks.



B blocks.



S-the permutation of L at the end of block sorting.



S' - sorted L.

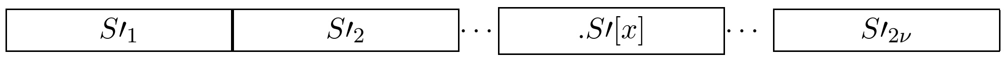


Figure 4

The arbitrary location of a value in L at the end of block sorting and at the end of the merge.

If $S[y]$ is no more than $k-1$ locations above its final sorted location, then at the end of a local block merge of any two adjacent blocks, any value in the first of these two adjacent blocks is in its final sorted location provided that all values in the previous block to the left adjacent block are already in their final sorted locations.

The reader should note that a block swap between A and B only occurs whenever the two values a and b are such that $A[a] > B[b]$ and the blocks in which a and b occurs are out of sequence in L by their mark. We need not consider the case where $A[a] = S'[x]$ and $A[a-1] = S'[x-1]$ (or $B[b] = S'[x]$ and $B[b-1] = S'[x-1]$) in the above proof.

Lemma 1 *If for any three adjacent blocks in S, the set of data values in the first block are sorted relative to the values in the next two blocks, then there are at least k values in the last two blocks that belong to the next k final sorted locations in the 2nd block.*

The proof of this lemma follows from the above invariant property and is not given here.

Proposition 1 Stable block sorting is sufficient to guarantee that Kronrod's and related algorithms will merge properly in the presence of more than k duplicate keys in a list.

Consider the set of duplicate values to be indexed according to their locations in A and B. If we use the indices of these repeat values to mark them as unique and as the relative value to any other repeat value of the same kind when it is compared during the merge operation, then block sorting in this way is equivalent to sorting where all values are unique. Therefore, stable block sorting will ensure that the invariant property proven in section 2 is maintained at the end of block sorting. Note that if a given value is repeated in the A list and the B list then we index repeat values in the B list with larger indices than those in the A list.

3.4 Correctness of Huang & Langston's Algorithm

It is easy to show that Huang & Langston's method correctly merge whenever the block sorting phase is implemented in a stable manner. Stability is not a correctness issue whenever the values in L are unique. Let $S_1 = S[x : x+q]$ and $S_2 = S[y : y+p]$ be two series in S. Where S is block sorted L and $0 \leq x \leq k, k \leq q \leq n-k, 0 \leq y \leq k-1, 1 \leq |S_1| \leq n-1, |S_2| = k-1$.

The values in S_1 and S_2 are in sorted order and therefore constitute a pair of natural runs in S. In addition, the values in these runs are in relative sorted order to all other values in S except for the last unmerge value in S_2 that remains at the end of the series merge. The remaining values in S_2 becomes the first block in the next series. A buffer of size k is sufficient to merge S_1 and S_2, since $|S_2| \leq k$. In addition, there is at least 1 buffer element to the front of series 1 during the merge. This element is at the location where the next selected value from the merge is to be place. Hence, the output values are always place in their correct final sorted location.

3.5 Sufficient Condition for Proper Merge

A sufficient condition for a correct merge is that at the end of block sorting, no value should be more than $k-1$ locations above their final sorted location.

This condition is satisfied at the end of block sorting for the case where there are no more than k duplicates of a given value. If there are more than k duplicates of a given value, then we have the following problems:

(1) The merge may actually not take place if the block sorting phase is unstable

(2) The overall merge is unstable even when the block sorting phase is stable and the values in the buffer are unique.

Permutations that can result in unstable merge even when the block sorting phase is stable is dealt with in [22, 23]. We see that stable block sorting does not necessarily result in stable merge even when a buffer consisting of unique values is used. A rigorous explanation on stable merge as a consequence of block re-arrangement is given in [22].

3.6 Counting Inversions in L and S

We state the following facts:

Fact 4 *The maximum locations that a value $L[r]$ can be above or below its final sorted location is $n-1$, where $0 \leq r \leq n-1$; i.e. if $L[r] = S\prime[t]$ then r and t satisfy the condition: $0 \leq |r-t| \leq n$.*

It is readily seen that there is a maximum of $n(n-1)$ inversions in L before block sorting. In this case, we assume that $m = n$. At the end of block sorting we have the following property on L as defined in theorem 1 below.

Theorem 1 *The maximum values in S_n belonging to $S\prime_{2n}$ or that is in S_{2n} but belongs to $S\prime_n$ is $k-1$.*

Proof If the set of values $L_n[r, r+1, \cdots r+t]$ are finally located in $S\prime_{2n}$ or $S\prime_n$ where $0 \leq t < n-1$. Since these values are sorted, they form a set of contiguous full blocks in L if $t > k$. In addition, there is only one block in $L_n[r : r+t]$ with mark less than $S\prime_{2n}[0]$ if $mod(t,k) \neq 0$. Since $mod(t,k)$ can assume a maximum value of $k-1$, then the argument for the proof is concluded. Similarly, there is a set of corresponding values in S_n adjacent to each other but belonging to $S\prime_{2n}$. These values create a maximum of $n(k-1)$ inversions in $S\prime$. We could have used the proof of this theorem to show the correctness of Huang & Langston's algorithm. Instead, we used a simple argument base on runs. We make the following observation. Since any element is no more than $k-1$ locations above its final sorted location at the end of block sorting, the maximum number of inversions in S is $n(k-1)$ and the maximum position of any given value below its final sorted location is n. However, the number of elements at maximum position below their final sorted location in n is less by a factor of $(k-1)/n$ in S.

4 Stable Block Sorting Techniques

Traditionally, block sorting is implemented in algorithms that uses block rearrangement using selection sort. However, selection sort is unstable and therefore, block sorting is unstable. We present four relatively simple and robust stable in-place block sorting techniques. The first of these techniques results in no more than $O(n)$ extra comparisons and data moves. The last two of these techniques results in no more than $n + O(\sqrt{n})$ extra swaps. The $3rd$ algorithm uses $O(n)$ bit level operations and no extra comparisons, while the $4th$ method uses $O(\sqrt{n}\lg(n))$ extra comparisons and swaps along with n extra swaps. Here we assume that $n = m$. None of our block sorting scheme uses an extra buffer. The first scheme uses block encoding and is limited by the number of unique values in L. Our second scheme is relatively simple and uses bit modification which is not affected by the presence of multiple types of duplicate values. The third scheme uses $O(\lg(m+n))$ extra bits to encode the sorted location of each block. In this scheme we set $k = (m+n)/\lg(m+n)$. The permutation cycles that determines the final location of each block are encoded and used during the block sorting operation. The first scheme is general and is independent of the block size whereas the third scheme is applicable where the block size is $O((m+n)/\lg(m+n))$. In our fourth scheme we also encode permutation cycles using $O(\sqrt{m+n}\lg(m+n))$ extra comparisons and swaps. In this case we set the number of blocks $\nu \approx k$. Since permutation cycles are used to implement block sorting, we see that $\approx \nu - 1$ block swaps is adequate to place these blocks in their final sorted locations. Therefore, block sorting requires no more than $O(m+n)$ extra swaps apart from those smaller order terms depending on the scheme that is used. This takes the total number of swaps to $2(m+n)$ for scheme 3 and $2(m+n) + O(\sqrt{m+n}\lg(m+n))$ for scheme 4. These schemes are describe below.

4.0.1 Scheme 1 - Duplicate Encoding

We use this scheme where the desire is to distinguish blocks with duplicate mark. It is easy to encode information that distinguishes each block as unique inside t adjacent blocks having the same mark. This is done by identifying the first t unique data values in L and then swapping them in sorted order with the second value in each of these block. The second to last value is swapped when the mark is taken as the last value in each block. These values are inserted back into their original locations when a block is place in its correct location or at the end of block sorting. For example, a binary search is used to locate each duplicate value at the end of block sorting. Since duplicate values are swapped in sorted order, they are also located and replace in sorted order. Using appropriate selection technique, it is only necessary to encode this information in blocks that have duplicate marks in A.

Blocks in A with duplicate marks are adjacent to each other. In addition, there may be more than one set of blocks with duplicate marks. A set of adjacent blocks in A consisting of t duplicate marks is encoded before the start of the block sorting phase as follows. A check is done to determine blocks that have equal marks. After locating these t adjacent blocks, a search for t consecutive unique values in L is done. The search for unique values starts at the second location in A if the mark of the first duplicate block is above the first block in A and terminate with the last value in the adjacent block below the first duplicate block. Otherwise, the search starts with the first unique value that follows the last duplicate value in block t. This search begins at the first unique location which follows the last duplicate value in block t. These t unique values are swapped with the data values at the second location in each of the t blocks. Searching and swapping of these t unique values is done sequentially. We do not swap a unique value if it happens to be a block mark or if the value is at a second location in a block that have duplicate values of a different kind. In the case where only the mark in block t is a duplicate value, the second value in this block is swapped with the kth value in the previous block. Of course, we do not confine the search for unique values to the A list. During the merge, the correct block among two blocks having equal mark is stably selected by comparing the second value in each block. The block with the smaller value in its second location is selected as the next block.

The encoded values are return to their original blocks in sorted order at the end of block sorting or once a block is selected into its correct sorted location. This is done by first locating duplicate block marks and then by doing a binary search on the other blocks for the corresponding duplicate value followed by a linear search of L to locate the insertion point for the second value in each block that have duplicate mark. A binary search is used to locate the first duplicate value apart from the sequence of duplicate values that is kept intact at the end of block sorting. No more than $O(\nu)$ extra operations is required for block encoding and decoding during this stable block sorting operation. In the case where only the mark in block t is a duplicate value, the $2nd$ value is restored as with other blocks followed by a swap with this value and the kth value in the previous block. Locating duplicates of a given kind is done during the local block merge operation. When the number of duplicates of a given kind in A exceed k, then we start counting to determine the value of t for this duplicate type. We terminate counting of blocks when the duplicate type changes.

4.0.2 Scheme 2 - Block Encoding by Bit Modification

Alternatively, if the data representation on the computational model been used allow for key modification and the merge operation is performed on record keys, then the $s = \lceil \lg(t) \rceil$ most-significant-bits (msb) are used in the data value at the $2nd$ location of each block to encode the original order in which these blocks occur in the sequence. Hence, the s msbs in the data value at the $2nd$ location of the first such block is set to 0_{10}, the s msbs in the $2nd$ such block is set to 1_{10} and so on. In the case where block t consist of only 1 duplicate value, a swap is made between the $2nd$ value in this block and the kth value in block $t-1$. The s msbs in the second value of this block is then set to $(t-1)_{10}$. During the block sorting phase, whenever 2 blocks of equal marks are selected, a comparison is done between the $2nd$ values in each block to resolve this clash. The block with the smallest $2nd$ value is selected as the next block. The $2nd$ value in an encoded duplicate block is restored whenever the block is place at its final sorted location. The information needed to restore the $2nd$ value is obtained from the $1st$ or any other duplicate value in each block. It is easy to determine that a selected block is the final block in the t sequence of blocks even though they may have been permuted during the sorting process. If during the comparison to select the previous block there was an equality but comparison to select the current block does not result in an equality when it is selected as the next block, then this block must be the last block in the series of t blocks. In this case, a comparison is done to determine if this block consisted of only one duplicate value by comparing the kth value in the previous block to any other duplicate value in the series. In the case where this block consists of only one duplicate value, the $2nd$ value is restored by the appropriate bit operation and a swap is done between this value and the kth value in the previous block. This method uses $< k$ comparisons and $O(2^t)$ bit operations. This approach is recommended where key modification is allowed. Note that, in the case where the kth value in each block is chosen as the mark, the data value at the $(k-1)th$ location in each block is used to encode the block sequence information.

In scheme - 1, if the last block have only 1 duplicate value as its mark, then the search for unique values begins at the $3rd$ location in this block if there are no unique values below the t duplicate block mark. In addition, this value is replace by a value that is larger than the value that was replaced in the other $t-1$ blocks.

4.0.3 Scheme 3 - Block Permutation Cycle Encoding

If we use blocks of size $k \approx n/c\lg(n)$, assume that the number of blocks to be sorted is strictly $c\lg(n)$. Therefore, we use an array of $c\lg(n)$ bits to encode the sorted location of each block. We use a *Block Encoding* word consisting of $c\lg(n)$ bits to encode the final sorted location of a block, where $c \geq 1$ is some constant. The final position of the mth block in sorted L is encoded by setting the mth bit in the Block Encoding word to 1 if this block comes from B otherwise this bit is set to 0. This encoding is done during the local block merge operation and therefore does not need the use of extra comparisons. This is possible because we keep a count of merged blocks as the local block merge proceed and we also note the list in which each merged block occurs. We use the Block Encoding word to encode the order in which blocks are to be selected to create a stable block sort as follows. It may be useful for the reader to go forward in the text and do some reading on our merge algorithm in order to have a clearer understanding of the way this encoding technique works.

(a) Two pointers are used to indicate the marks of the A and B blocks to be used in the next comparison. If the block from A has the smaller mark or is equal to the mark in B, then the first bit position in the Block Encoding word is set to 0 else the first bit position is set to 1. The appropriate pointer is incremented by k to point to the mark in the next A block if the selected block is from the A list. Otherwise, the pointer is incremented to point to the mark of the next B block. A second comparison is done to determine the next block. If this block comes from A, the next bit position is set to 0 else this bit is set to 1. Continue in this manner until all blocks are selected and their corresponding bit position set to 0 when a block from A is selected or is set to 1 when a block from B is selected. At the end of this process, the $c\lg(n)$ bits that constitute the Block Encoding word will encode the sequence in which blocks from A and B are to be selected to create a stable block sort. The bit pattern created from this operation encodes the permutation cycles that are used to place the blocks in their sorted position in a stable manner. These bit positions can be simultaneously set with our local block merge operation. Therefore no extra comparisons need to be done to create this encoding word.

(b) During block placement, we also use a second array of $c\lg(n)$ extra bits to encode information that indicates when a block has been place in its final sorted position. We call this the *Block Placement* word. The first bit in the block encoded word encodes the list and the block location in this list from which the first block originates to create the block sorted sequence. During the block sorting operation, if the first bit in the block encoded word is set to 1. Then we know that the first sorted block comes from the first B block. Therefore, we swap the $1st$ block in L with the first block in B. We then determine the block that belongs to the first block location in B. In our case, the first B block location is at $\nu - 1$ since we have decided that the B list should begin at the original buffer location at the end of the local block merge operation. To determine the correct block that belongs to location $\nu - 1$, we do a scan of bit location $\nu - 1$ in the block encoded word to determine the bit type. If the bit type is 0, then we count the number of 0's up to this bit position to determine the block location in the A list. This is the block that is to be placed in block location $\nu - 1$. We then do appropriate block swapping. On the other hand, if this bit is 1, we count the number of 1's up to this bit position to determine the block location in the B list and the do appropriate swapping. We continue placing blocks in this fashion until we have completed a block permutation (placement) cycle. A block placement cycle completes whenever the $1st$ block in a cycle reaches its final sorted location.

At the start of each block placement cycle, we determine the correct location of the first block that is swapped as follows. Starting at the first bit location in the Block Encoded word, we count bits of the type at this location until the bit position that correspond to this block location is reached. Let the number of such bit location be m, where $2 \leq m \leq c \lg(n)$. Therefore, the final destination of this first block is at the mth block position in block sorted L. Therefore, during our block placement cycle operation we check to determine if the next block to be placed is at this location. If this is the case, then we know that our block placement cycle is complete and a new cycle should begin.

The Block Placement word encodes blocks that are already place in their final sorted location during the block sort operation. Originally, these bits are all set to 1s. When a block is selected into its final sorted location, the corresponding bit is set to 0. Therefore, we know where to start a new block placement cycle by doing a check to determine the next bit that is set to 1 in this array. A new block placement cycle starts with the block in this position in L. Note that we set all bit positions that correspond to singleton cycles to 0 before we begin the block placement operation. The block placement operation ends whenever the Block Placement word is finally set to 0_{10}.

4.0.4 Scheme 4 - Block Cycle Encoding In The Presence of $\leq k/2$ duplicates

Before using this method we ensure the following 2 conditions are satisfied.

(1) The number of element in each block $k \approx \nu = \lceil\sqrt{2n}\rceil$, where ν is the number of k size blocks in L.

(2) The maximum duplicate of any given value in L does not exceed $\frac{1}{2}k$.

Since our merge algorithm creates two sorted list from A and B in the locations $L[0 : m-(k+1)]$ and $L[m-k : m+n-(k+s)]$ in block size units then we can encode the actual location where a sorted block should be place during the block sort phase of our merge algorithm by making use of the ordered information in each merged block. Let the number of unmerge blocks in the list previously occupied by A excluding the buffer be ν_1 and the number of blocks occupied by the second list be ν_2. We have $\nu = \nu_1 + \nu_2 \approx k$. We keep a counter q whose value at any given time is the block number been created from a local block merge operation. During our series merge operation, at the end of merging the jth block in the first or second list of L, we swap the value at the jth location with the value at location q. If $j \neq q$ then these two values are out of sort and are used to encode the correct sorted location of block q in sorted L. During our block sorting operation, we simply check the jth location for an out of sort data value. If an out of sort data value exist, then we know that this block is not placed in its final sorted location. Therefore, we do a binary search operation to locate a second out of sort value in this block. The location of this second value encodes the final sorted block location for this block. The block at the location to be swapped is checked in a similar manner to determine its correct sorted location, after which the two blocks are swapped. The two out of sort values are then returned to their correct sorted location by a single swap operation. We continue in this fashion until a cycle of out of sort blocks are properly placed. We search for a new cycle by checking the adjacent block location where a previous cycle started. In addition, we keep a count of all placed blocks. The placement operation ends when this count equals ν.

Computation to determine the second out of sort value is done using $O(lg(k))$ comparisons. Hence, we see that we need at most an extra $\nu \lg(\nu)$ comparisons and ν swaps to implement our block sort stably by this method. This $\nu \lg(\nu)$ extra comparisons is less than the standard $O(\nu^2)$ required by selection sort. Careful consideration of scheme 1 above reveals how this method can be easily modify to implement stable block sorting in the presence of more than k duplicate values in L. Hence we use a maximum of $\nu(1+\lg(\nu))$ comparisons and $(\nu-1)k-(2s+h)+\nu-1$ swaps.

5 A Practical In-Place Merge Technique

We now describe an unstable algorithm that does optimum in-place merge. We use example illustrations to give a clear understanding of how the algorithm works. We then explain the modifications needed to
(1) Reduce the number of comparisons and moves and
(2) Make the algorithm stable. In essence, the algorithm works by first doing stable local block merge on the values in A and B followed by block rearrangement. It is possible to use an internal buffer to merge A and B in this manner because of the following fact.

Fact 5 *If L is sorted and then block permuted so that the values in A and B remain sorted, any stable block sorting algorithm can be used to regain sorted L in a stable manner. In addition, the value at any location within a given block can be used to select that block during the block sorting.*

The merge is performed by first extracting a buffer as described above. The location block merge begins with the first buffer location. At the end of the local block merge operation, the values in L are sorted but permuted in k size blocks. Therefore, a stable block-sorting algorithm is used to rearrange the blocks to create a sorted sequence.

5.0.5 Buffer Extraction

We use pointers $p1$ and $p2$ to locate the k largest values in L as follows. Set $p1$ and $p2$ to point to $A[m-1]$ and $B[n-1]$ respectively. Compare the two values pointed to. Decrement the value of the pointer by 1 that points to the larger of the two values. Compare the next two values and repeat until the k largest values are selected in this manner. At this time, the k largest values in L are $A[p1 : m-1]$ and $B[p2 : n-1]$, where $m-k \leq p1 \leq m-1, n-k \leq p2 \leq n-1$ and $m+n-(p1+p2+1) = k$. Figure 5 illustrates the way in which $p1$ and $p2$ are used to determine the locations of the k largest values in L.

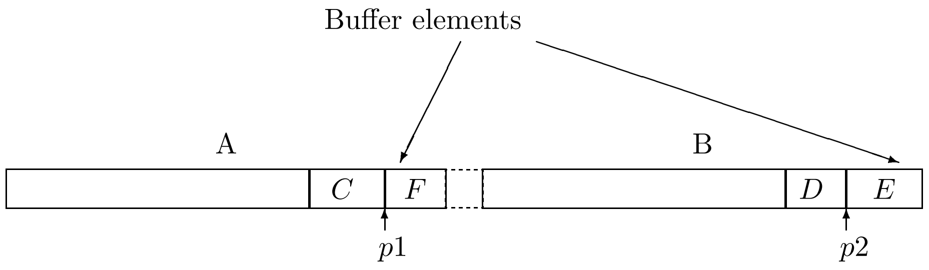


Figure 5
The k largest values in L are located in blocks E and F pointed to by pointers $p1$ and $p2$. Buffer size as illustrated are exaggerated for the purpose of emphasis and clarity of explanation.

Denote the values $A[m-k : p1]$ and $B[n-k : p2]$ as C and D respectively. Note that $|C| = m-p2$. We bring the buffer into 1 k size block at the end of A whilst at the same time merging the values in the undersize block C with the remaining non-buffer elements in B by using E as an output buffer as follows. Note that $C[0]$ forms the mark of the last k size block in A and therefore we have $|C| = |E|$. We merge by comparing the value $A[p1 - 1]$ in C to the value $B[p2 - 1]$. The larger of the two values goes into the highest location in E. We decrement appropriate pointers and continue to merge in this manner until the values in C are fully merge with the remaining values in B. At this point, the h largest values in L occupies the h highest locations in B, where $|D| \leq h \leq n - |C|$. At this point, the originally sorted buffer elements from block E are now located in block C but possible permuted.

Buffer extraction uses $k - 1$ comparisons. Merging blocks C and D uses $h - 1$ comparisons and h swaps. This brings the total number of comparisons at this stage to $k + h - 2$ and the total swaps to h. Where h is the actual number of values merged at the end of the B list.

5.1 Ensuring Integer Number of Unmerge k Size Blocks

We ensure that the last value in the last k size block in B is $\geq$ the last value in the last block in A. Therefore, we would normally prefer that $h \geq s = mod(n, k)$. If the last merged value in B falls above the last k size boundary, here we count k size blocks starting at the first location in B, then we merge the remaining unmerge $(s - h + 1)$ values in the s highest locations of B with the non-buffer elements in A. We use the last $(s - h + 1)$ locations of the buffer as output. Our objective here is to ensure that our local block merge operation always conclude with exhaustion of the A list. We then swap the $(s-h+1)$ merged values at locations $L[m + n - (s - h + 1) : m + n - (h + 1)]$ in the buffer with the buffer elements at locations $L[m+n-(s+1) : m+n-(s-h+1)]$. We use $2(s-h+1)$ data moves to implement this block permutation instead of data swapping. Because some buffer elements may have rotated at the end of the A list during this merge operation, we do another block rotation to restore the buffer to the last k locations in A. We mark the last merged value at the end of A to determine when the local block merge operation is to be terminated. Possible configuration of L at the end of this stage is illustrated by

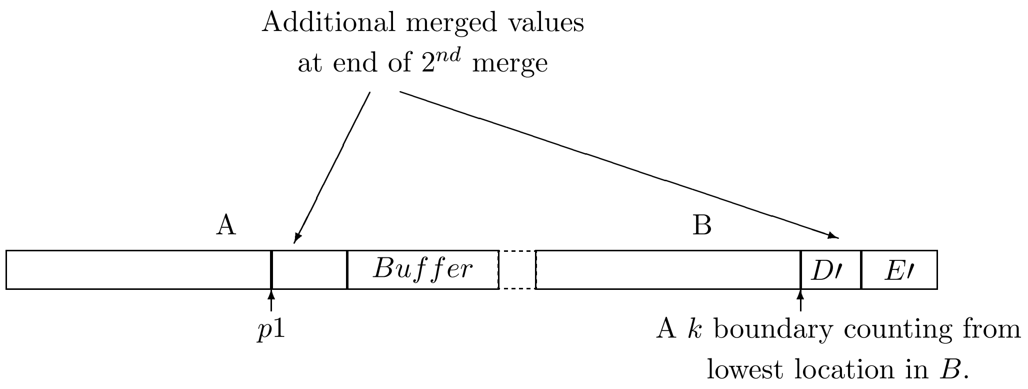

Additional merged values at end of 2nd merge
A
B
Buffer
D′
E′
p1
A k boundary counting from lowest location in B.

Figure 6
Configuration of L at the end of the first merge phase.

Let the number of merged values at the end of the A list be $h\prime$. For the case where our block size is set to $\frac{m+n}{\lg(m+n)}$, if $h\prime$ lies below a given number of k size boundary, then we set the appropriate bit position in our Block Encoding word to ensure that these blocks are place in their proper sorted locations during the block sorting phase of the algorithm. In addition, we terminate the merge operation after the next merged value in the A list is the last unmerged value.

Because our technique does local block merge before block sorting, it is easy to use a more efficient alternative approach as follows. We use pointer $p2$ to mark the location of the last merge value at the h highest locations in B. Therefore, the last block to be merged from the B list may be an undersize block. The location of the last element in this block been the location pointed to by $p2$. Similarly, we use pointer $p1$ to mark the last merge value in A. Therefore the merge operation terminates when either of these values is reached.

5.2 Placing the Leftmost s Smallest Values in L

The $s\prime$ highest locations in the buffer are used to merge the values at the $s\prime$ lowest locations in A with the remaining unmerge values in B, where $s\prime = mod(m, k)$. The objective of this merge operation is to place the $s\prime$ smallest values in L at the $s\prime$ smallest locations in A. This operation uses a minimum of $s\prime - 1$ comparisons and $s\prime$ swaps. Another $s\prime$ swaps at the end of this phase bring the total number of swaps to $2s\prime$. The extra $s\prime$ swaps are used to place the $s\prime$ smallest values at the $s\prime$ lowest locations in A whilst at the same time restoring the buffer to a single k size block. Again, restoring the buffer can be done by using straight data moves instead of data swaps. We use $2s\prime + 1$ data moves to achieve this. Note that a sequence of the next r smallest values, where $0 \leq r \leq n - (s + 1)$ may also be merged into the head of the B list during this merge operation. At this point, the remaining unmerge values in L are located within an integer number of k size blocks apart from the blocks in B that contain the r additional merged values. The last merged value in A is at a location above the last location that has an unmerge value in this list. Similarly, the first unmerge value in B is at a location pointed to by R. Therefore, we see that this operation requires a maximum of $r + s\prime - 1$ comparisons and $r + s\prime$ data swaps. Another $s\prime$ swaps at the end of this phase bring the total number of data swaps to $r + 2s\prime$. Where the values at the first r locations in B are the next r smallest values in L. Figure 7 illustrates the possible configuration of L at the end of this phase. The number of unmerged values excluding the buffer is $m + n - (k + s + s\prime + r)$. At this point, we are now ready to determine pairs of unmerge full blocks in L and then merge them using the buffer as output.

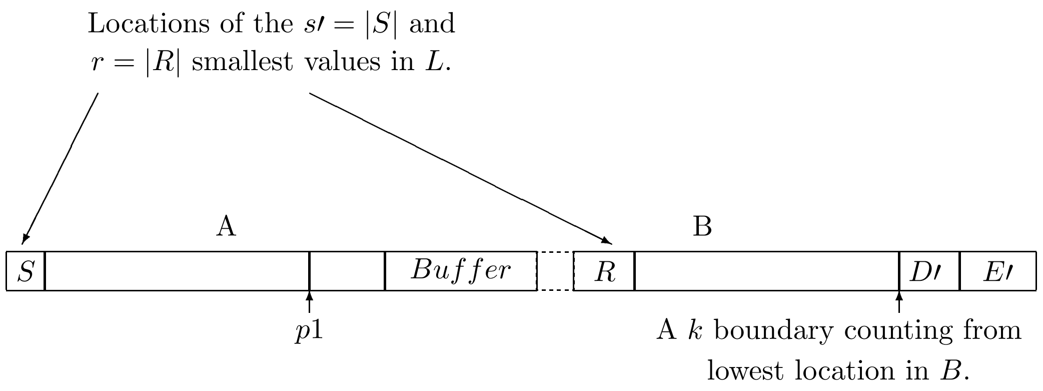

Locations of the $s\prime = |S|$ and
$r = |R|$ smallest values in L.
A
B
S
$Buffer$
R
$D\prime$
$E\prime$
$p1$
A k boundary counting from
lowest location in B.

Figure 7
Location of the $(s\prime + r)$ smallest values in L at the end of the 3rd merge phase.

5.3 Determining the next output buffer location

At the end of merging the first k values into the original buffer location, the buffer may be split into two parts. The buffer is displaced into locations previously occupied by values that are merged. We determine the first location that is the next k size buffer location that will hold the next merged full block. This is done by comparing the marks of the two current blocks. Recall that a current block is a block on which a merge operation is currently been done. Figure 8 illustrates a possible sequence of configurations of L during and at the end of the local block merge operation. The value at the last location in each block is chosen as the mark. The block with the smallest mark is determined. The lowest buffer location to the front of the block with the smallest mark is the next output buffer location. In the case where two blocks have the same mark, the block from the A list is chosen as the next block. In addition, during the merge operation, we give values from the A list first preference in the case of a tie. This ensures that the merge operation at this point is actually stable. No more than $\nu - 1$ comparisons is used to locate each block during this phase of the merge.

5.3.1 Merging Pairs of Blocks

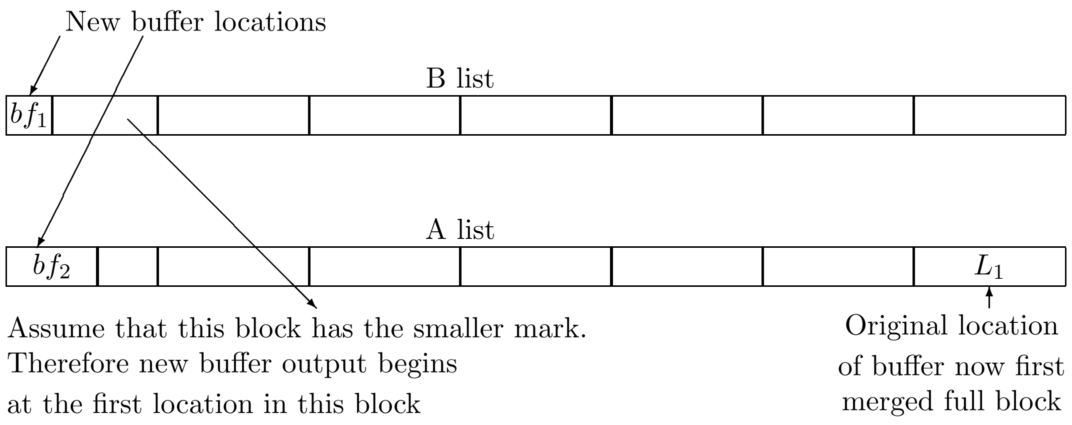
Assume that number of unmerged full blocks in A is the same as the number of unmerged full blocks in B. Also use the last value in each block as their mark. A possible sequence of merged blocks in L and the positions of the buffer is illustrated in figure 8.

5.3.2 An Example Instance of Merging

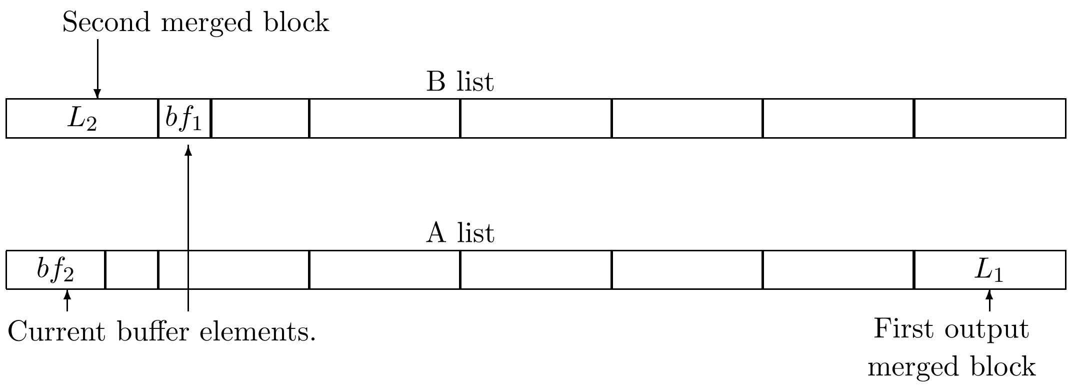
The example below illustrates the operation of the merge algorithm.

We use the undersize blocks bf_1 and bf_2 to represent the location of the buffer at any given moment during the local block merge operation. The ith merged block in L is represented as L_i. An unmerge block is represented as the original block fromm the A or B list.

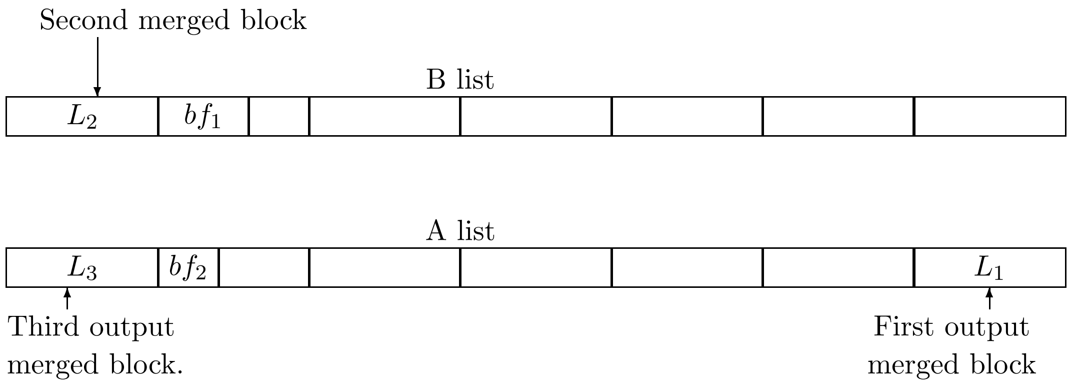
(1) At the end of merging the first block, a possible configuration of A and B is illustrated below.


New buffer locations
B list
bf_1
A list
bf_2
L_1
Assume that this block has the smaller mark.
Therefore new buffer output begins
at the first location in this block
Original location
of buffer now first
merged full block

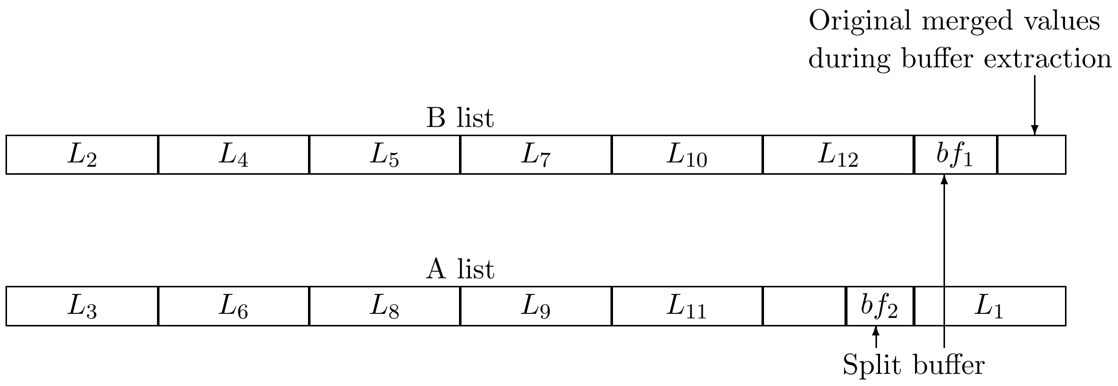
(2) Possible distribution of values in A and B at the end of merging the 2nd block in L.



(3) Assume now that A_1 is the next block with the smallest mark. The distribution of values in L at the end of merging L_3 is as illustrated below.



Possible permutation of blocks in L at the end of merging all blocks



Note that except for the last k size block in A and the permuted buffer, the values in both lists are sorted. A single block exchange operation brings the buffer into a single k size block whilst at the same time bringing the two undersized merged blocks into a single full block. The buffer is then sorted using an optimum sort algorithm. Block sorting on L excluding the buffer, creates a single sorted list as desired.

Permutation of L at the end of block sorting.

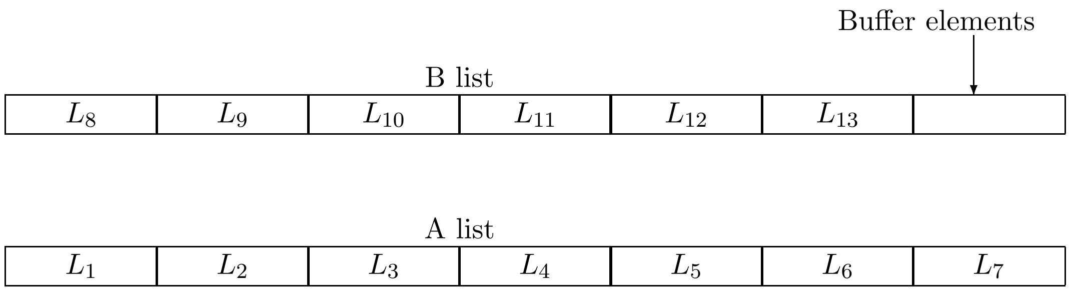


Figure 8
A possible sequence of local block merge operations for given input permutation of A and B.

5.4 Merging Two Current Blocks

At the end of merging a full block, the buffer may be split into two parts. Each part is located to the front of the current blocks. At this point, the algorithm is ready to merge the next full size block. At the end of merging k values from the two current blocks, a single sorted sequence is created that occupies previous locations of the buffer. See figure 8. When the buffer is split into two parts during the merge of two current blocks, output from the current merge is place in these buffer locations until a full block of merged values in obtained. In this way, it is seen that a sequence of merge values always start and end at k size boundaries. These locations been the same as those in the previous unmerge permutation of L.

We see that at the end of merging all blocks, the block with the rth smallest mark always contain the $(rk + s)th$ smallest values in L, where $0 \leq r \leq 2\nu - 1$. This invariant property makes it possible to use a simplified version of scheme 1 given above to implement a stable block sort on L at the end of merging all blocks. Effectively, the merge is complete when an output value is either the last unmerged value at the end of A or B. Since before the start of the local block merge we had permuted L so that the merge will terminate on the exhaustion of A. We do appropriate swapping of remaining data values in the k size block in B that forms the last selected block in B with remaining data values in the buffer at the end of A to bring the buffer into 1 piece.

We do a block transformation to set the buffer above the h or s merged values at the end of B. At this point, we have done a maximum of $(\nu - 1)$ comparisons to locate blocks for merging in sorted order, a maximum of $m+n-(k+s+s\prime+r-1)$ comparisons and $m+n-(k+s+s\prime+r)$ swaps during the local block merge operation and an extra $2b+1$ moves to restore the buffer to a single k size block, where b is the number of buffer elements in A at the end of the local block merge. In addition, we use a further $k+h+1$ moves to place the buffer in its correct location at the end of merging. We use the algorithm given in [3] for permuting the location of the buffer.

We sort the values in the buffer by doing $O(\sqrt{m+n}\lg(m+n))$ or $O((m+n)(1-\frac{\lg\lg(m+n)}{\lg(m+n)}))$ comparisons and $O(\sqrt{m+n}\lg(m+n))$ or $O((m+n)(1-\frac{\lg\lg(m+n)}{\lg(m+n)}))$ swaps. At this point, the values in L are sorted in non-decreasing order.

5.5 Maximum Number of Comparisons and Moves

Summeries from the tabulation below gives us total number of comparisons $m+n+(\nu-1)$ (or $m+n+\nu(\nu+5)-6)$ $+o(m+n)$, swaps $2(m+n)$ $+o(m+n)$
and $2(k+s+s\prime+r)+1$ moves.
Summery totals for comparisons, swaps and moves are tabulated in table 3 below. In this tabulation we use block size $\frac{m+n}{\lg(m+n)}$ with $\lg(m+n)$ extra bits for block placement cycle encoding.

Table 3 Summary analysis of dynamic Buffer merge

	COMPARISONS		DATA MOVES	
$STEP$	Minimum	Maximum	Minimum	Maximum
1	$k+h-2$	$k+h-2$	h	h
2	$s-h+h\prime-2$	$s-h+h\prime-1$	$2(s-h+1)+h\prime$	$2(s-h+1)+h\prime$
3	$r+s-1$	$r+s-1$	$r+2s$	$r+2s$
4	$m+n+\nu-s-$ $k-h-r-h\prime$	$2(\nu-1)+m+n-$ $s-h-h\prime-k-r$	$m+n+2\nu$ $-s-k-r$	$m+n-s-h-k-$ $r-h\prime+2(\nu-1)$
5	$O(1)$	$O(1)$	$4(m+n)+$ $2(\nu-s-k)$	$(m+n)-$ $(2s+k)$
6	$o(m+n)$	$o(m+n)$	$o(m+n)$	$o(m+n)$
	$\approx m+n+\nu$ $+s-h+o(m+n)$	$=m+n+2\nu+s$ $-h+o(m+n)$	$\approx 5(m+n)$ $-6k+o(m+n)$	$\approx 6(m+n)+$ $3(\nu-1)+o(m+n)$

Complexity analysis sumerise in table 3 above is done by taking the following optimizing technique into account.

5.6 Optimizing the Merge Operation

We reduce the average number of data comparisons and data moves during the local block merge operation as follows.

(1) By making better use of the two pointers $p1$ and $p2$ as follows. Use $p2$ to indicate the location of the last merged value at the end of B when merging the h or s largest values in L. Similarly, we use the pointer r to indicate the last merged value at the head of the B list when merging the $s+r$ smallest values in L. Therefore, we move the r smallest values into the buffer and locate the next k boundary after the $(r-1)th$ merged value in B. The buffer rolls in front of B until the first unmerge value is reached. Use the first value located above the last buffer element in B as the first value in this first undersize block in B. Therefore, number of remaining values to be merge is $(m+n)-(s+r+k+h+h\prime)$.

Output during the first paired series merge operation starts at the first buffer location which is k locations below the rth location in B.

(2) Whenever the last value in a newly selected block to be merged is less than the first value in the second current block, there is no need for comparisons and swaps to create the next merged full block in L. Instead, all data values in that block are moved into the buffer by doing $2k + 1$ data moves. This operation is possible because at least 1 buffer element is always below the elements to be merged.

(3) Since buffer elements are permuted during the local block merge operation, we use $2(m + n - (k + s + s\prime + r)) + 1$ moves to merge instead of $(m + n - (k + s + s\prime + r))$ swaps. We show how this is done further on.

(4) If a selection sort is used to do block sorting at the end of merging L then it is possible to use $2(m + n - (k + s + s\prime + r)) + 1$ moves to place blocks in their sorted locations instead of using $(m + n - (k + s + s\prime + r))$ swaps. We show how this is done below.

We implement these operations by first extracting the $k + 1$ largest values in L into the buffer. Therefore, the $k + 1$ largest values in L are located at the $k + 1$ highest locations in A. We then merge the $1st$ value in A with the smallest values in B using the last buffer location. A buffer element is now at $A[0]$ and the smallest value in L is at $A[m - 1]$. Therefore, we reset the last k size boundary in A at location $A[m - 2]$. We also decrement the location of all k boundaries in B by 1. We then merge the $s\prime = mod(m - 1, k)$ values at locations $A[1 : s\prime]$ with the remaining unmerge values at the head of the B list as follows.

(4.1) Shift the already merged $r\prime$ values into the $r\prime$ locations starting at $L[m - s\prime]$ by a block permutation operation. Therefore, last buffer element is at location $L[m + r\prime - 1]$.

(4.2) Merge the $s\prime$ values at $A[1 : s\prime]$ with $B[r\prime : n - s - 1]$ using buffer elements at locations $L[m + r\prime - s : n + r\prime - 1]$ as output locations.

(4.3) Exchange buffer elements at locations $A[0 : s\prime - 1]$ with the next $s\prime$ smallest values at locations $L[m - (s\prime + 2) : m - 2]$. Use data moves employing buffer elements to implement these block exchange. The buffer is now in 1 piece at the last k size block in A with a single buffer element at the head of A. This element constitute our hole to be used for effecting data moves during block sorting. Recall that the k boundaries for elements in A and B have been moved back by 1 element. We use a block permutation operation to move the merged values at the head of B to the front of the buffer elements whilst at the same time moving the buffer elements adjacent to the first unmerge values in B. We then proceed with the local block merge in the usual way.

5.7 Implementation Results

Results from experimental test for the case where $k = \lfloor\sqrt{m+n}\rfloor$ and a selection sort is used to implement the block sort as described above are tabulated below. In this case, results are place in columns headed $Alg1$. We also list results for the case where $k = (m+n)/\lg(m+n)$ with block sorting implemented by use of $O(\lg(m+n))$ extra bits, we place these results in collumns headed $Alg2$. Figure 9 tabulates our experimental results for implementation using block size $O(\sqrt{m+n})$ and $O(\frac{m+n}{\lg(m+n)})$ for $Alg1$ and $Alg2$ respectively. Tabulation in figure 9 assumes that $m = n$ and does not include buffer sorting. Block sorting was implemented using the selection sort. This adds an extra $\nu(\nu+1)/2 \approx 0.5n + O(\sqrt{(}n))$ comparisons to our merge algorithm.

	Compares(C)		Moves(M)		C/n		M/n	
n	$Alg1$	$Alg2$	$Alg1$	$Alg2$	$Alg1$	$Alg2$	$Alg1$	$Alg2$
25	69.30	56.60	220.00	210.00	2.77	2.26	8.80	8.40
50	78.80	104.80	462.80	453.10	1.58	2.10	9.26	9.06
100	288.00	199.50	822.20	1056.40	2.88	1.99	8.22	10.56
250	771.90	513.50	2019.30	2163.90	3.09	2.00	8.08	8.66
500	1506.30	1015.20	4007.20	4505.00	3.01	2.05	8.01	9.01
1000	3054.90	2017.40	7963.40	8772.40	3.05	2.03	7.96	8.77
2500	7448.40	5020.50	19964.80	22942.10	2.98	2.02	7.99	9.18
5000	15110.50	10022.20	39989.20	46117.000	3.02	2.01	8.00	9.22
10000	29899.30	20023.70	79873.40	90598.80	2.99	2.00	7.99	9.06
20000	57990.90	40025.80	163842.70	190802.60	2.90	2.00	8.19	9.54
25000	75120.90	50026.70	199702.41	237631.80	3.00	2.00	7.99	9.51
35000	105154.30	70028.10	279463.19	300909.20	3.00	2.00	7.98	8.60
45000	135155.09	90028.40	359327.41	386233.59	3.00	2.00	7.99	8.58
50000	150399.20	100027.50	399509.91	428899.80	3.01	2.00	7.99	8.58
				Averages	2.89	2.03	8.16	9.05

Figure 9

Experimental results from implementation of our algorithm.

6 Conclusions And Remarks

We have done a detailed analysis of Kronrod's and Huang & Langston's merge algorithms. We have confirm the importance of stability for the correctness of these algorithms whenever there are more than k duplicates of a given value in the input lists. Our complexity analysis is supported by results from experimental implementations. We also showed four novel techniques for doing stable block sorting in-place that can be used to make these algorithms stable. In our implementation, we avoided the problem of correctness due to unstable block merge by doing local block merge in the natural way. Therefore, we have simplified the problem of stability to that of buffer extraction. A trade off from this approach is a reduce number of data moves and data comparisons.

References

[1] Aho, A. V. ,Hopcroft, J. E. and Ullman, J. D. .
The Design and Analysis of Computer Algorithms.
Addison-Wesley, 1974.

[2] Carlsson, S.
Splitmerge-A fast stable merging algorithm
Information Processing Letters
22 (1986) 189-192,

[3] Ching-Kuang Shene
An Analysis of Two In-Place Array Rotation Algorithms
The Computer Journal
40 #9 (1997) 541 - 546,

[4] Dewar, R. B. K.
A stable minimum storage algorithm
Information Processing Letter
2 (1974) 162-164

[5] Dudzinski, K and Dydek, A
On Stable minimum Storage Algorithm
Information Processing Letters
12 (1981) 5-8

[6] Dvorak, S. and Durian, B.
Stable linear time sublinear space merging
The Computer Journal
30 (1987) 372-375

[7] Dvorak, S. and Durian, B.
Unstable linear time O(1) space merging
The Computer Journal
31 (1988) 279-282

[8] Dvorak, S. and Durian, B.
Merging by decomposition revisited
The Computer Journal
31 (1988) 553-555

[9] Hovarth, E. C.
Stable sorting in asymptotically optimal time and extra space,
Jour of the ACM
25 (1978) 177-199

[10] Huang. B.C. and Langston, M. A.
Practical In-place Merging.
Communications of the ACM
31 #3 (1988)

[11] Huang, B.C. and Langston, M. A.
Fast Stable Sorting in Constant Extra Space.
The Computer Journal
35 #6 (1992) 643-649

[12] Huang, B.C. and Langston, M. A.
Stable duplicate-key extraction with optimal time and space bounds.
Acta Informatica
26 (1989) 473-484

[13] Hwang, F.K. and Lin, S.
A simple algorithm for merging two disjoint linearly ordered sets.

SIAM Journal on Computing
1 #6 (1972) 31-39

[14] Kemp, R. and Teubner, B.G.
Fundamentals of the Average Case Analysis of Particular Algorithms.
John Wiley & Sons, 1984

[15] Knuth, D. E.,
The Art of Computer Programming; C-3 Sorting and Searching
Addison-Wesley, Reading, MA 1973

[16] Kronrod, M. A.,
Optimal ordering algorithm without operational field,
Soviet Mathematics,
10 (1969) 744-746

[17] Mannila, H.
Measures of Presortedness And Optimal Algorithms.
IEEE Transaction in Computing
34 (1985) 318-325

[18] Mannila, H. and Ukkonen, E.
A simple linear-time algorithm for in situ merging,
Information Processing Letter
18 #4, (May 1984) 203-208

[19] Mohammed, John L. and Subi, Carlos S.
An Improved Block-Interchanged algorithm
Journal of Algorithms
8 (1987) 113-121

[20] Munro, J. I. and Raman, V.
Sorting with minimum data movement

Journal of Algorithms
13 (1992) 374-393

[21] Munro, J. I. and Raman, V.
Selection from read-only memory and sorting with minimum data movement
Theoretical Computer Science
165 (1996) 311-323

[22] Pardo, L. Trabb
Stable Sorting and merging with optimal space and time bounds,
SIAM Journal of Computing
6 (1977) 351-372

[23] Salowe, J. and Steiger, W.
Simplified Stable Merging Tasks,
Journal of Algorithms
8 (1987) 557-571

[24] Symvonis, A.
Optimal Stable Merging,
The Computer Journal
38 #8 (1995) 681-690

[25] Valliant, L. G.
Parallelism in Comparison Problems,
SIAM Journal of Computing
4 #3 (1977) 348-355

[26] Geffert, V. , Katajainen, J. and Panasen, T.
Asymptotically Efficient In-place Merging
Theoretical Computer Science
237 #1-2 (2000) 159-181