Joint Conference of 1996 International Computer Symposium
December 19~21, Kaohsiung, Taiwan, R.O.C.

Interleaved Mergesort

Shih-Hung Lin and Jiang-Hsing Chu
Department of Computer Science
Southern Illinois University at Carbondale
Carbondale, IL 62901, USA

Abstract

We present a new mergesort algorithm which is based on partitioning the input list in an interleaving manner. The new algorithm has a better performance than the classic mergesort algorithm does. On the average, the new algorithm runs in $O(n \log n)$ time using $O(\sqrt{n})$ extra space, where n is the size of the array to be sorted. In the worst case, depending on the implementation, the new algorithm may have $O(n \log n)$ runing time using $O(n)$ extra space, which is the same as the classic mergesort, or it may have $O(n\sqrt{n})$ running time using $O(\sqrt{n})$ extra space.

1 Introduction

Sorting has involved human life ever since the invention of digits. It is also one of the most intensively studied subjects in computer science[4]. People in this field have been trying to find a "better" sorting algorithm, even though it has been proven [1, 8] that $O(n \log n)$ is the lower bound to the complexity of any sequential comparison-based sorting algorithms, where n is the size of the list of data elements. In the past few decades, many sorting algorithms achieved this lower bound. One among them is Mergesort[1].

Mergesort employs the divide-and-conquer technique. It consists of two phases, *split* and *merge*. In the split phase, the input list is divided into smaller sublists. These sublists are eventually merged in the merge phase after being sorted by the same method. Both top-down (recursive) and bottom-up (non-recursive) approaches can be used to implement mergesort. In the classic mergesort, the input list is divided into smaller contiguous segments. We will refer to the classic mergesort as the *segmented mergesort* hereafter.

The merge operation is the heart and soul of mergesort. One obvious way to merge two sorted sublists is to compare the leading elements of the two lists and move the appropriate one, the smaller one for non-decreasing or the larger one for non-increasing order, to a new location. The index to the removed element is updated to the next element in its list. Given two lists with a total size of n, this mechanism ensures that every element relocates orderly in a time complexity of $O(n)$. It is straightforward but an extra space of size $O(n)$ is required [4].

In-place merge algorithms which merge two sublists using an extra space of a constant size have been developed [5, 2]. Although these algorithms achieve the theoretical lower bound to sorting, they are are not practical because of large overheads. Huang's algorithm [2] is a more practical method, but it is still about 2 to 3 times slower than the classic merge algorithm described above.

Our goal is to find a trade-off between time and space, i.e., to find an algorithm with a running time comparable to the classic mergesort algorithm, without the requirement of the $O(n)$ extra space. We discovered that if the input list is divided in a different manner, interleaved instead of segmented, it is possible to merge two sublists in-place with help of a queue, whose size depends on the distribution of the input data.

In this paper, we will present a new sorting algorithm named *interleaved mergesort* based on this discovery. We will also give an analysis of interleaved mergesort. It turns out that interleave mergesort can sort a size n list in $O(n \log n)$ time in most cases and uses only an extra space of size $O(\sqrt{n})$.

2 Interleaved Mergesort

Interleaved mergesort applies the divide-and-conquer technique as well to partition the input list in an *interleaving* manner instead of segmentation. An *interleaved sublist* out of a regular list is defined by specifying its *list-leader* and *offset*. A list-leader is the first element in its sublist. Every two successive elements in an interleaved sublist are separated by a fixed number of elements from the list. The distance between two successive elements is called the offset.

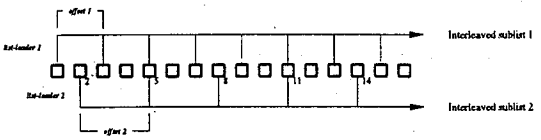

Figure 1: Interleaved Sublists

As shown in Figure 1, the interleaved sublist 1 consisting of elements in positions $1, 3, 5, \ldots, 15$ is an interleaved sublist whose list-leader is 1 and offset is 2. Similarly, the interleaved sublist 2 consisting of elements in positions 2,5,8,11, and 14 is an interleaved sublist whose list-leader is 2 and offset is 3.

In interleaved mergesort, each sublist is an interleaved list whose offset is a power of 2. These sublists are sorted separately and merged in pairs into sorted interleaved lists with a smaller offset, half of that in

sorting algorithm spares is

$$S(n) = \sum_{k=1}^{\log_2 2n} \frac{3}{2}\sqrt{2^{k-1}}\pi\frac{n}{2^{k-1}} = \frac{3}{2}n\sqrt{\pi}\sum_{k=0}^{\log_2 n}\frac{1}{\sqrt{2^k}}$$

and

$$S(n) - \frac{1}{\sqrt{2}}S(n) = \frac{3}{2}n\sqrt{\pi}\left(\frac{1}{\sqrt{2^0}} - \frac{1}{\sqrt{2n}}\right).$$

When n is large, $1 \gg \frac{1}{\sqrt{2n}}$. Therefore,

$$S(n) \approx \frac{3}{2}n\sqrt{\pi}\frac{1}{1-\frac{1}{\sqrt{2}}} = \frac{3}{2-\sqrt{2}}n\sqrt{\pi} \approx 9.0773n,$$

which is $O(n)$, where $n = m/2$ and m is the size of data to be sorted.

If the sorted interleaved sublists are large and random enough, there are lots of locations where the domination status reverses in every iteration. The number of standing elements is which may not be relatively large to the data size. When all the savings of merge operations in all iterations add up, the total becomes $O(n)$, which is substantial.

3.4 Worst Case: Time and Space

We know interleaved mergesort avoids some data movements for those elements who initially are already in their final positions before merge operation. In our analysis, we already considered the data comparisons and movements for the queue operations. However, we did not take into account the extra data comparisons and movements incurred when the queue is full and an overflow-handling procedure is invoked. The worst case time and space complexities of interleaved mergesort depends on the overflow-handling procedure used,

Suppose we use the first overflow-handling procedure, discussed in the earlier section, where when the queue is full, items are merged back with the sublists. Obviously, the worst case space requirement is $O(\sqrt{n})$ since no additional space is allocated. The cost of overflow-handling is proportional to the elements left. Since each time the queue is full, at least $\sqrt{n}$ elements must have been output to their final locations. The worst case time complexity is

$$(n - \sqrt{n}) + (n - 2\sqrt{n}) + \ldots,$$

which is $O(n\sqrt{n})$.

In the second overflow-handling procedure discussed earlier, we simply double the size of the queue whenever it is full. Obviously, the worst case space requirement is $O(n)$ since we need space to hold the whose sublist. There are extra data movements required to move elements from old queue to new queue. In the worst case, the number of such data movements is

$$\sqrt{n} + 2\sqrt{n} + 4\sqrt{n} + \ldots,$$

which is $O(n)$. Therefore, the overall worst case time complexity is still $O(n \log n)$.

4 Empirical Results and Observations

In order to observe and compare the behaviors of segmented and interleaved mergesort, two C programs were written; one for segmented mergesort and the other for interleaved mergesort. Each of them was tested on a Sun SPARC5/110mhz with 32 MB of main memory. They are capable of sorting arrays of up to 5,000,000 long integers. The users can specify the size of array for both programs. The programs will report the numbers of comparisons and data movements involved, and the execution time. In interleaved mergesort, the program also requests the size of queue and reports how many times the queue becomes full. The results from our experiments are given in the following sections.

4.1 Effect of Queue Size on Performance

We calculated $prob_{safe}(n, k\sqrt{n})$ and noticed that greatly increasing the factor k will not increase the probability much. In practice, a large factor k may not improve the performance but waste space. The following table is an experimental result of sorting 2,000,000 keys using interleaved mergesort. The largest scale of merging in this case is merging 2 lists of 1,000,000 keys. The result consists of using different sizes of queue, from 10% to 160% of $\sqrt{1,000,000}$.

There is a characteristic of the results that should be noticed. The number of operations is strongly related to overflow-handling procedure which is called every time when queue is full. In each trial for the same size of queue, those larger numbers of operations are always associated with larger numbers of times when queue is full. Those extra operations come from the overhead doing overflow-handling procedure. The results are plotted in Figure 8.

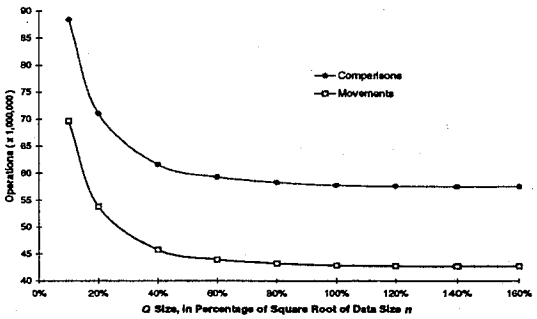

Figure 8: Number of Operations vs. Size of Queue

The graph shows that both numbers of comparisons and data movements decrease as the size of queue increases. Increasing the size of queue benefits the performance by reducing the operations. However, the decreasing rate of operations becomes very slow, when the size of queue is larger than square root of n, 1,000. It is even harder to tell if the size increase

states L_2 dominates L_1. Those bold edges by which a shaded node transits to another shaded node represent the transitions which reverse the domination status. There exists a standing element every where a bold edge is used. Accordingly, the number of spared data movement equals 3/2 times the expected number of bold edges used by a random path.

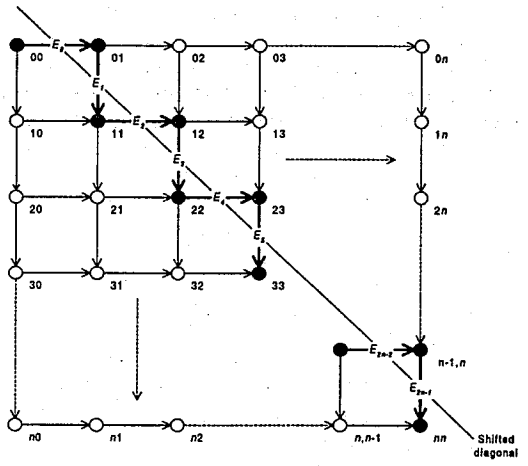

Figure 7: Traveler's Lattice for Standing Elements

In order to figure out the expected number of bold edges used by a random path, we categorize the bold edges into two types, left-to-right and top-to-bottom. Let N_{xy} denote the node labeled xy (or x, y to avoid ambiguity). We defined the bold edge from N_{xy} across the diagonal as E_k with $k = x + y$, where $0 \leq k < 2n$. From the diagram, we see an edge E_k is horizontally from $N_{k/2,k/2}$ to $N_{k/2,k/2+1}$ if k is an even; and vertically from $N_{(k-1)/2,(k+1)/2}$ to $N_{(k+1)/2,(k+1)/2}$, otherwise.

Note that the total number of paths from N_{ab} to N_{cd} (with $a \leq c$ and $b \leq d$) is

$$\binom{(c-a)+(d-b)}{(c-a)}.$$

Define $U(E_k)$ as the number of paths, from N_{00} to N_{nn}, which use E_k. Every such path starts from N_{00} and reaches the node before E_k, then passes through E_k and continues its way to N_{nn}. Such a path can be broken into two sub-paths, before and after using E_k, connected by E_k. Therefore, $U(E_k)$, is the product of the number of sub-paths before using E_k and the number of sub-paths after using E_k.

Given an E_k, we have

$$U(E_k) = \binom{k}{k/2}\binom{2n-(k+1)}{n-k/2} \text{ if } k \text{ is even}$$

$$U(E_k) = \binom{k}{(k-1)/2}\binom{2n-(k+1)}{n-(k+1)/2} \text{ if } k \text{ is odd}$$

Let $E(s)$ denote the expected number of standing elements, which is also the expected number of E_k's used by a random path from N_{00} to N_{nn}. We have

$$\begin{aligned} E(s) &= \left(\sum_{k=0}^{n-1}(U(E_{2k}) + U(E_{2k+1}))\right) \Big/ \binom{2n}{n} \\ &= 2\left(\sum_{k=0}^{n-1}\binom{2k}{k}\binom{2n-2k-1}{n-k}\right) \Big/ \binom{2n}{n} \end{aligned}$$

Since [8]

$$\sum_{k=0}^{n}\binom{2k}{k}\binom{2n-2k}{n-k} = 4^n$$

and

$$\binom{2n-2k-1}{n-k} = \frac{1}{2}\binom{2n-2k}{n-k},$$

we have

$$\begin{aligned} E(s) &= 2\left(\frac{1}{2}\sum_{k=0}^{n-1}\binom{2k}{k}\binom{2n-2k}{n-k}\right) \Big/ \binom{2n}{n} \\ &= \left(4^n - \binom{2n}{n}\right) \Big/ \binom{2n}{n} \\ &= 4^n \Big/ \binom{2n}{n} - 1. \end{aligned}$$

From [3], we know that

$$\binom{2n}{n} \approx \frac{2^{2n}}{\sqrt{n\pi}}.$$

Hence, we determine that the expected number of standing elements is

$$E(s) \approx 4^n \Big/ \left(\frac{2^{2n}}{\sqrt{n\pi}}\right) = \sqrt{n\pi},$$

where n is number of elements in each list to be merged. Therefore, the expected number of data movements which a single merge operation can spare is $3\sqrt{n\pi}/2$, which is $O(\sqrt{n})$.

In interleaved mergesort, the algorithm merges different sizes of lists in different iterations. There are $\lceil \log_2 m \rceil$ iterations, where m is the size of the input list to be sorted. For simplicity, let's assume that $m = 2n > 0$ and n is a power of 2. There are $\log_2 2n$ iterations. In the first iteration, there are n merge operations each of which merges lists of 1 elements. In the following iteration, the number of merge operations becomes half of that in the previous iteration, and each of them merges lists of as twice elements as previous. In the kth iteration, there are $n/2^{k-1}$ merge operations, and each of which merges lists of 2^{k-1} elements. Therefore, the total of data movements our

less than the average height of Catalan binary trees. Therefore, we can conclude that the average usage of the queue is $O(\sqrt{n})$.

Although we can not calculate the exact order of $q_{avg}(n)$, we know it is bounded above by $\sqrt{n}$. In practical use of interleaved merge, the probability of *not* using overflow-handling procedure to handle overflow is more concerned, for a given size of Q. It is defined as the following.

$$prob_{safe} = \frac{path(n,q)}{path(n,n)},$$

where n is the size of one list and q is the size of Q.

We may wonder if really a size $\sqrt{n}$ queue is large enough. In Figure 6, the curve of $prob_{safe}(n, k\sqrt{n})$ is plotted for 10 different k's. The probability of *not* using overflow-handling procedure increases when the factor k increases. For those curves with smaller k's, they oscillate violently and drop very fast when $n < 50$. As n increases, the oscillation becomes smaller and each of the curves tends to be bounded by a smaller range. On the other hand, those curves with larger k's are more stable and become bounded very fast. This indicates that the probability depends on the factor k and it is irrelevant to n if n is relatively large.

If we draw a horizontal line at each of the constant values, we will find that the density of these lines become higher as the k increases. From $k = 1.00$ to $k = 2.00$, the probability increases drastically from 40% up to roughly 98%. In order to increase the probability to 99.99% however, the factor k should be increase to 3.00. The probability does not increase as drastically after k is larger than certain value.

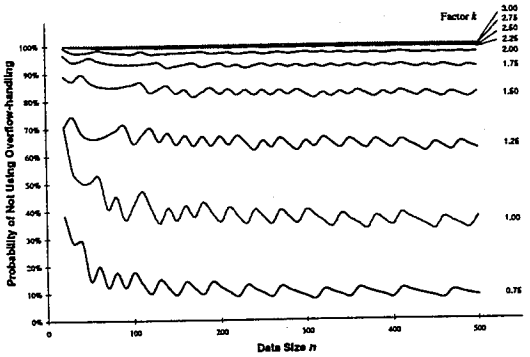

Figure 6: Choices between Time and Space

3.3 Time: Average Data Movements

It has been proven that the lower bound to merging two sorted lists of sizes m and n is $m+n-1$ comparisons in the worst case [4]. In segmented mergesort, a merge operation always takes place between two adjacent contiguous sublists. An obvious way to merge is to compare their leading elements and output the appropriate one to an extra space at a time. The comparison repeats until running into the end of one list. The rest of the other list will be appended to the output. The output should be copied back to the original space. Given two adjacent subarrays, L_1 followed by L_2 of sizes m and n respectively, this scheme requires $2(m+n)$ data movements and $m+n$ extra space for the output array.

There is a better approach in which the first subarray L_1 is copied to a temporary array T, which is then merged with L_2. The result is written to the space originally occupied by L_1 and L_2. Recall that L_1 and L_2 are adjacent. Some elements in L_2 need not be moved in the case when T runs out of elements before L_2 does. This scheme needs $2m+n$ data movements in the worst case and an extra space of size m.

Our new algorithm, interleaved merge, still requires $m+n-1$ comparisons in the worst case. However it has fewer data movements and uses less extra space in most cases. The data movements can be categorized according to the direction to which the elements are relocated, namely *standing*, *forward*, and *backward*. No movement is required for those elements at their final locations initially. Only one step is required to move an element forward to its final location. Those that need to be moved backward require two steps, one to the queue and the other to their destinations.

To compute the total number of data movements, let s, f, and b be the numbers of elements categorized as standing, moving forward, and moving backward respectively, where $s+f+b = m+n$. From the flowchart in Figure 3, an element needs to be moved backward for every element which is moved forward, that is, $f = b$. No movements are needed for the standing elements. Each forward element requires only one movement. However, each backward element needs two movements, one into the queue and the other out of the queue. The whole process requires a total of $3b$ data movements. In comparison with the old algorithm's $2m+n$ movements, the new algorithm reduces $2m+n-3b$ movements. Assume $m = n$ for simplicity. Since $m+n = s+f+b$, we have $2n = s+2b$, which yields $3n = 3s/2+3b$. The number of spared movements is $2m+n-3b = 3n-3b = 3s/2$. The new merge algorithm reduces the number of data movements by $3s/2$.

The above result does not show how good the improvement is. It depends on the permutation of the input list. Wherever the domination status reverses, there must be at least two standing elements. In order to know the expected number of standing elements, we need to figure out the expected number of times the domination status reverses using a traveler's lattice.

Suppose we are merging two interleaved sublists L_1 and L_2, each of size n, and the index of L_1's list-leader is less than L_2's. The n by n traveler's lattice shown in Figure 7 imitates the situation. The main diagonal is shifted as before. From any node below the diagonal, a transition transferring to its bottom increases the cost while to its right decreases the cost. Conversely, from any node above the diagonal, a transition transferring to its bottom decreases the cost while to its right increases the cost. The shaded nodes beside the diagonal, such as $00, 01, 11, 12, 22, \ldots$, are the states without using the queue. All the nodes above the shaded nodes represent the states L_1 dominates L_2, while those below the shaded nodes represent the

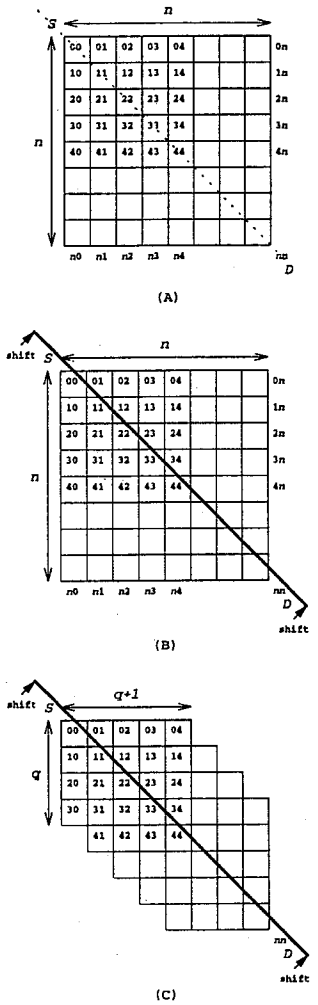

Figure 5: Traveler's Lattices

When two interleaved sublists are merged, the result of comparisons in the process is a winning-losing sequence. In each comparison, the traveler moves to the next vertex on its right if the leader of L_1 wins, to the one below otherwise. Once the traveler reaches the right border, all elements in L_1 have arrived at their final locations. All the remaining elements in L_2 will be appended to the end without further comparisons. Similar situation happens if the traveler reaches the bottom border. After $2n$ steps, the traveler will arrive at D.

The cost of a given path can be defined as the maximum distance the traveler deviates from the main diagonal for a given path. It also represents the usage of the queue for the given sequence. One position in the queue is consumed if the traveler gets farther from the main diagonal while one position is released if the traveler gets closer to the main diagonal.

This model is slightly different from interleaved merge due to the asymmetry in interleaved merge. For example, the merge process which corresponds to the path 00, 01, 11, 12, 22, 23, ..., nn has a cost 0 while the merge process which corresponds to the path 00, 10, 11, 21, 22, 32, ..., nn has a cost 1. In order to correct the difference between these two models, the main diagonal is shifted half width of a cell as shown in Figure 5(B). This shift breaks the symmetry of the diagram and correctly models interleaved merge.

3.2 Space: Average Queue Usage

As stated earlier, interleaved merge without overflow handling needs an extra space of size n in the worst case, where n is the size of the larger sublist. No extra space is needed in the best case. It does not have a better space utilization than the classic mergesort if we have to provide enough space for the worst case. Given the fact that the possibility of using a space near the size for the worst case is tiny, we should use a smaller queue and call a procedure to handle overflow occasionally. In this section we will discuss the possibility of overflowing a given queue.

With a slight modification, the traveler's lattice model helps us investigate the probability of overflowing a given queue Q. Suppose the size of queue is q, with $q \leq n$. Paths are *valid* if their costs are not higher than q. Any path with a cost higher than q is called an *invalid* path. An invalid path corresponds to a sequence which will overflow the queue. As shown in Figure 5(C), the model can be modified by removing some cells to simulate this situation. Any path which uses only vertices in Figure 5(C) has a cost less than or equal to q.

These valid paths correspond to all the possible winning-losing sequences which do not overflow the queue when merging two lists. We define the number of valid paths as a function of n and q, $path(n, q)$.

From [6, 7], we have

$$path(n,q) = \sum_{i=0}^{n/s} \left(2\binom{2n}{n-is} - \binom{2n}{n-(q+1)-is} - \binom{2n}{n-(q+2)-is} \right) - \binom{2n}{n},$$

where $s = 2q + 3$, n is the size of the sublists to be merged, and q is the size of extra memory allowed.

Suppose we are merging two lists of size n. The number of paths with a cost k is $path(n, k) - path(n, k-1)$. Therefore, the average usage of Q can be written as

$$q_{avg}(n) = \sum_{k=1}^{n} \frac{k\,(path(n,k) - path(n,k-1))}{path(n,n)}.$$

Unfortunately, we are unable to simplify this formula any further. Nevertheless, in the Catalan model [8] of binary trees, each binary tree corresponds to a path traveling on the upper-right half of a traveler's lattice. It has been proven that the average height (corresponding to cost in our case) of Catalan binary trees is $O(\sqrt{n})$. In our model, paths may cross the main diagonal, which allows more lower cost paths to exist. As a result, the average cost in our model should be

When Q is not empty, comparisons only happen between the leader of the dominating list and the front of Q since the leader of the dominated list has been moved into the front of Q. If the element at the front of Q is less than the leader of the dominating list, it will be removed from Q and moved to its final location. Otherwise, the current leader will be moved to the location after the element occupying it has been moved into Q. The process will repeat until one of indices reaches the end of the list. The remaining elements in Q and the dominated list will then be attached to the tail of the sorted list.

In the best case, all elements are initially in their final positions. They merge like closing a zipper. No data movement is required so Q is never used. In the worst case, when one sublist totally dominates the other, Q requires positions that equal the size of the dominated list. When a list is dominated, we move some of its elements into Q in order to preserve them and let the other list move into its space. When the situation reverses, move the elements in Q out to their final positions and free the space in Q. No matter which list dominates, the other uses Q as a buffer. An example of interleaved merge is shown in Figure 4.

Later in our analysis, we will show that to sort a list of size n, a queue of size $O(\sqrt{n})$ is enough most of the time. However, a queue of size $O(n)$ is needed in the worst case. There will be no saving in extra space if we provide with a queue of size $O(n)$. Therefore, in our algorithm, we will only use a queue of size $O(\sqrt{n})$, and a employ overflow-handling procedure when the queue is full. There are several overflow-handling procedures that can be used. We outline two of them below.

In the first procedure, the space allocated for Q is never changed. When Q is full, we split Q into two interleaved sublists and merge them with the remaining of L_1 and L_2, respectively, using the classic merging technique and reuse their free locations from avail to $index_1$ and $index_2$. After merge, $index_1$ and $index_2$ are adjusted to include those elements from Q, and Q becomes empty again. The interleaved mergesort continues its task from here.

The second procedure simply allocates more space for Q when needed. Depending on the memory management routines provided by the programming language chosen to implement the algorithm, it might be necessary to copy the contents in Q. If this is the case, increasing the queue size linearly will result in a total running time worse than $O(n \log n)$ in the worst case. However, if we double the queue size every time, the worst-case running time can be reduced.

The worst case running time and space requirement depend on which overflow-handling method is used. They will be discussed later.

3 Analysis

Clearly the core in mergesort is the merge operation. Our analysis of interleaved mergesort focuses on merge process first and then move on to the whole sorting algorithm. The number of operations and space utilization are the two major concerns in this analysis. We introduce the traveler's lattice model to simulate the problems.

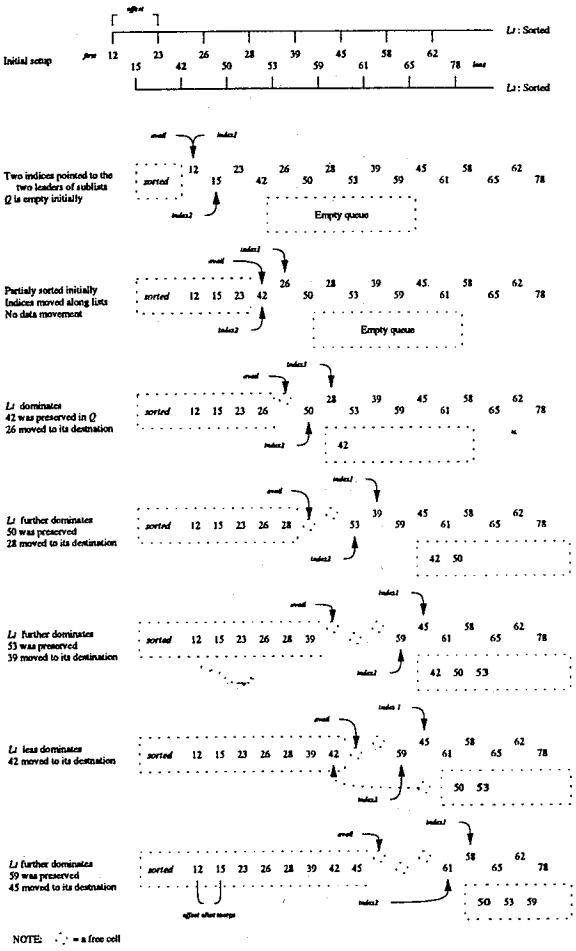

Figure 4: Data Movement in Interleaved Merge

3.1 Traveler's Lattices

Recall that if two interleaved sublists can be merged, their sizes should not differ by more than one. We therefore assume any two sublists to be merged have the same size for simplicity.

Suppose there are two sorted interleaved sublists L_1 and L_2 both of size n satisfying the conditions of interleaved merge. To merge them, comparisons only take place between their current leaders. Assume that the leaders of L_1 and L_2 have an equal chance of winning the comparisons. How often one list can dominate the other is similar to how far a random walker can travel away from the origin.

An equivalent model called *traveler's lattice* helps us analyze the time and space complexity of interleaved merge. Consider an n by n lattice diagram illustrated in Figure 5(A) as a di-graph with only top-to-bottom and left-to-right edges. A vertex is labeled xy (or x, y to avoid ambiguity) if it is x units away from the left border and y units away from the top border. A traveler travels from $S(00)$ to $D(nn)$.

the original sublists. The input list becomes sorted when there is only one sublist left.

Figure 2 illustrates the idea of interleaved mergesort. Using the definition given earlier, L is an interleaved list with an offset 1. Interleaved mergesort views L as an interleaved list and partitions its elements to form two interleaved sublists, L_1 and L_2. The first one, L_1, starts at the first element of the original list and the other, L_2, starts at one offset from the first element (i.e., the second element). The new offset of both sublists is twice of the original offset. By applying the same scheme on both L_1 and L_2 recursively, the list will finally becomes single elements before the second phase starts.

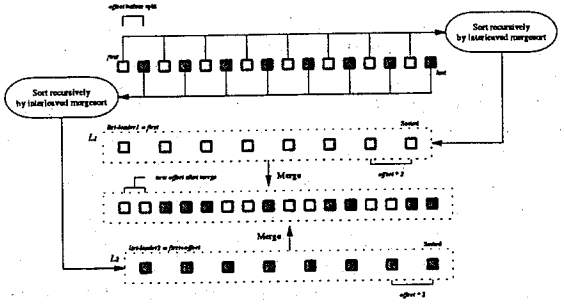

Figure 2: The Scheme of Interleaved Mergesort

This split scheme looks a little more complicated than that in segmented mergesort. However, it does not require any data comparison at all. This scheme does not produce any drag on speed in the split phase and benefits the second phase in terms of space utilization. The effect will be discussed in the later sections.

In order to be merged into a larger interleaved sublist which fits in the original space, two sorted interleaved sublists must meet the following criteria. They must have the same offset. The indices of their first elements must differ by one half of the offset. And their sizes should not differ by more than one. In the case when their sizes differ by one, the list-leader of the longer sublist must have a smaller index than the list-leader of the other sublist. For example, a possible partner to merge with an interleaved sublist with indices 1, 9, 17, 25 and 33 is the one with 5, 13, 21 and 29. Both sublists have the same offset 8. The difference between the first indices of lists, 1 and 5, is half of the offset. The first sublist has only one more element than the second sublist. They conform to all the conditions for interleaved merge.

These conditions may seem awfully difficult to meet. Fortunately, in each iteration of interleave mergesort, the two sublists generated in the split phase satisfy these conditions automatically and can be merged and stored back to their original space with the algorithm described below.

The flowchart in Figure 3 depicts the process of *interleaved merge*, which merges two interleaved sublists into a larger interleaved sublist in-place using a queue as a buffer for temporarily holding those elements that are not in their final locations. The following discussion assumes elements are to be sorted into non-decreasing order. For non-increasing order, it can be obtained similarly.

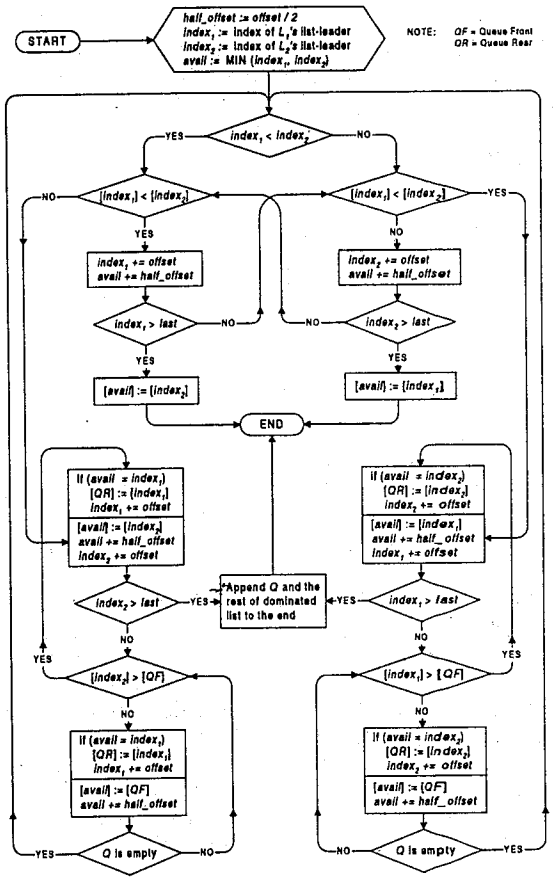

Figure 3: Flowchart of Interleaved Merge

In the merge process, two indices, $index_1$ and $index_2$, are used to point to the current leaders of L_1 and L_2 respectively, and the queue Q is used as a temporary storage. Initially, $index_1$ and $index_2$ point to the list-leaders of L_1 and L_2 respectively, and Q is empty. Another pointer, *avail*, points to the position available for next element to be moved in. It is assigned to the smaller value of $index_1$ and $index_2$ initially. It points to the next available location after all preceding interleaved elements have been merged.

There are three major regions in the flowchart. In the upper region, Q is empty and no data movement is required in this region. The lower-left and lower-right regions are for sending elements to their final locations and preserving data in Q before it is overwritten.

When Q is empty, $index_1$ and $index_2$ will move along L_1 and L_2 respectively and *avail* will follow if the elements pointed by $index_1$ and $index_2$ are in proper order. Otherwise, we say *domination* happens and Q comes to play. The sublist contains the smaller leader is called the *dominating list* and the other list is called the *dominated list*. The leader of the dominated list releases its position for the leader of the dominating list after it is moved to Q for preservation. The utilization of Q will increase in this case.

of queue still helps reduce the number of operations when queue is larger than 1,200 positions.

4.2 Interleaved versus Segmented

The following experiment is conducted to compare interleaved and segmented mergesort. Different sizes of data were sorted by both algorithms. The size of queue for interleaved mergesort was set to 120% of square root of n. The average of four trials is plotted in Figure 9.

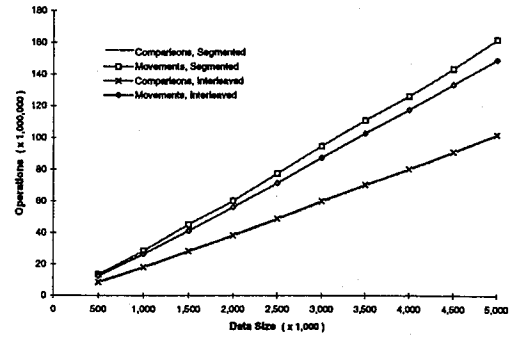

Figure 9: Operations, Interleaved vs. Segmented

The numbers of comparisons in the two algorithms are almost the same for any data size in our experiments. They are both slightly larger than twice of the data sizes. However, the number of data movements has a significant difference. The difference comes from those standing elements discussed earlier. Interleaved mergesort uses fewer data movements than segmented mergesort does to complete its tasks. The difference increases as the size of data increases. The execution time results are plotted in Figure 10.

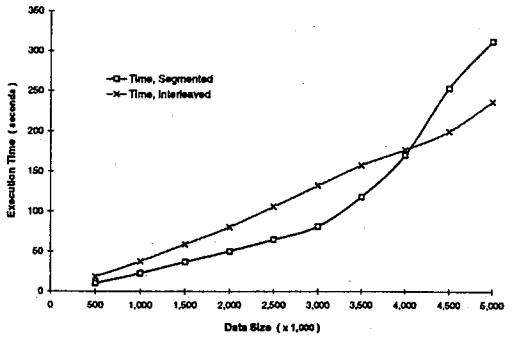

Figure 10: Performance, Interleaved vs. Segmented

For a small data size, interleaved mergesort spends more time than segmented mergesort does because interleaved mergesort has a larger overhead. As the data size increases, interleaved mergesort picks up its performance and catches up segmented mergesort when the data size larger than 4,000,000. There are two reasons for the this result. First, as the data size increases, the difference between the numbers of data movements in two algorithms increases. Interleave mergesort gradually overcomes its overhead and catches up segmented mergesort. Second, segmented mergesort uses much larger extra space than interleaved mergesort does. It will cause more page faults when the program running on an operating system using virtual memory, such as UNIX. The savings in both numbers of data movements and size of extra space attain this achievement.

5 Conclusion

When we sort data of large size, the execution time and the size of space required are the two major concerns. In this paper, we present a new algorithm which we call interleaved mergesort. The intention to design this algorithm was to provide a trade-off between time and space. The analysis and empirical results surprisingly show that it actually saves space without wasting time for large input.

The algorithm sorts elements in $O(n \log n)$ time which is the lower bound to the complexity of any sequential comparison-based sorting algorithms. To sort a large list using interleaved mergesort, the number of comparisons is the same as the classic mergesort, but the number of data movement is less. The analysis shows that the number of data movements which the new algorithm can spare is proportional to the data size. Although interleaved mergesort involves some overhead, it outperforms the classic mergesort by reducing the number of data movements when the size of input data is large. Moreover, it uses extra space only of $O(\sqrt{n})$ size which is much less than the extra space used in the classic mergesort algorithm.

The primary use of mergesort is for external sorting. It is not obvious how interleave sort can be adapted to an external sort. This will be an interesting topic for further study.

References

[1] S. Baase. *Computer Algorithms: Introduction to Design and Analysis.* Addison-Wesley, Reading, MA., 1991.

[2] B. Huang and M. Langston. Practical in-place merging. *Commun. ACM*, 31(3):348–352, 1988.

[3] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison-Wesley, Reading, MA., 1973.

[4] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, Reading, MA., 1973.

[5] H. Mannila and E. Ukkonen. A simple linear-time algorithm for insitu merging. *Information Processing Leters*, 18:203–208, 1984.

[6] S. G. Mohanty. *Lattice Path Counting and Application.* Academic Press, New York, NY, 1979.

[7] T. V. Narayna. *Lattice Path Combinatorics with Statistical Applications.* University of Toronto Press, Toronto, Ontario Canada, 1979.

[8] R. Sedgewick and P. Flajolet. *Analysis of Algorithms.* Addison-Wesley, Reading, MA., 1996.