Enhanced Merge Sort Using Simplified Transferrable Auxiliary Space

Zirou Qiu, Ziping Liu, Xuesong Zhang
Department of Computer Science
Southeast Missouri State University
Cape Girardeau, MO 63701
zqiu1s@semo.edu, zliu@semo.edu, xzhang@semo.edu

Abstract

This paper presents an enhanced implementation for the classic merge sort algorithm which reduces both the execution time as well as the extra space requirements. The main idea is to modify the merge procedure such that it only moves half of the input array into one single auxiliary array instead of the conventional whole array and two auxiliary arrays. Experimental data proves that the proposed implementation is faster than the conventional implementation.

Keywords: Merge sort, Auxiliary array, Time complexity, Space complexity

1 Introduction

Merge sort is a classic $O(n \log_2(n))$ algorithm of sorting elements externally using divide-and-conquer technique. It is faster than insertion sort at large input size [2] (book). However, due to the need of copying the entire input array into auxiliary arrays at each level of recursion tree, merge sort is troubled by its large constant factor [2] (book). Because of this dilemma, merge sort is slower than insertion sort at small input size. In practice, merge sort is outperformed by quick sort, which is an algorithm of sorting elements internally using divide-and-conquer technique [1] (journal).

Several enhancements from various aspects have been conducted by researchers aiming to improve the execution time of Merge Sort. In [3] (journal), Gupta found a way to improve the dividing procedures by splitting the input array into three or more subarrays and then applying parallel programming to solve each. In [6] (journal), Paira came up with a way to alter the merge procedure by grouping elements in pairs and then extracting smaller elements from each pair to form the left subarray. In [5] (journal), Renu reduced the execution time of Merge Sort by combining it with the Quick Sort. To eliminate the $O(n)$ space complexity, in-place merge sort was designed. However, the need of swapping elements frequently adds the burden of large constant factor on in-place merge sort and thus not effective in practice [4] (journal). Hence, in [4] (journal), Huang proposed a new method of in-place merge sort which guarantees the total number of exchange is bounded below by $3.5 * n$ where $n$ is the size of the input array.

In this paper, we introduce an enhanced merge sort implementation which adopts a new way of merging. It outperforms the conventional implementation for merge sort at any input size and reduces the execution time by up to 44.18% at input size of 15,000,000 compared with conventional implementation. Detailed discussions on our proposed implementation are given in the following sections. In section 2 our proposed algorithm is delineated with pseudocode. In section 3 the analysis of the proposed implementation is studied. And lastly section 4 presents the conclusion.

2 Proposed Implementation

The merge procedure of the conventional merge sort requires copying both left and right subarrays to two auxiliary arrays for merging each pair of subarrays. Our enhanced merge procedure copies only left subarray to an auxiliary array. It then uses the left subarray as the initial workspace. The algorithm compares the next elements in the auxiliary array with the next elements in the right subarray, and move the smaller one into the next element in workspace. Available slots in workspace decreases with elements moving into it and expands with elements moving out of right subarray.

The comparison between elements in two arrays stops when one of the two cases occurs: case 1, all elements in the auxiliary array have been moved back into the input array; case 2, all elements in the right subarray have been moved to somewhere in the input array. Two arrays are successfully merged when case 1 occurs. However, when case 2 is raised, there must exist element(s) in the auxiliary array that haven’t been copied back to input array. Our algorithm completes merging by moving them back in order.

2.1 Pseudocode of the Proposed Merge Procedure

This subsection starts with the pseudocode of the merge procedure for conventional merge sort algorithm. The pseudocode is from Cormen’s book [2] (book) and we call it Cormen’s merge. Then the proposed implementation of the merge procedure is presented in subsection 2.1.2 and we call it Qiu’s merge.
2.1.1 Cormen’s Merge Procedure of Conventional Merge Sort

Merge (A, p, q, r)
01 n1 = q - p + 1
02 n2 = r - q
03 let L[0..n1] and R[0..n2] be 2 empty arrays
05 FOR i = 0 to n1 - 1 DO:
05 L[i] = A[p+i]
06 END FOR
07 FOR j = 0 to n2 - 1 DO:
08 R[j] = A[q + 1 + j]
09 END FOR
10 L[n1] = positive infinity
11 R[n2] = positive infinity
12 i = 0
13 j = 0
14 FOR k = p to r DO:
15 IF L[i] <= R[j] DO:
16 A[k] = L[i]
17 i += 1
18 END IF
19 ELSE DO:
20 A[k] = R[j]
21 j += 1
22 END ELSE
23 END FOR

2.1.2 Qiu’s Merge Procedure of the Proposed Implementation

Note:
A: input array; s: starting index, m: middle index, e: ending index
By the nature of merge sort, two subarrays are sorted

Merge (A, s, m, e)
01 let a1 be an empty auxiliary array
02 FOR j1 = s to m DO:
03 append A[j1] into a1
04 END FOR
05 i = s
06 j = m + 1
07 k = 0
08 WHILE k < a1.size() && j <= e DO:
09 IF a1[k] <= A[j] DO:
10 A[j] = a1[k]
11 k += 1
12 END IF
13 ELSE DO:
15 j += 1
16 END ELSE
17 i += 1
18 END WHILE
19 WHILE k != a1.size() DO:

2.1.3 Merge Sort Procedure

Note: This procedure is unchanged for Qiu’s implementation.

MergeSort(A, s, e)
01 IF s < e DO:
02 m = (s + e) / 2
03 MergeSort(A, s, m)
04 MergeSort(A, m+1, e)
05 Merge(A, s, m, e)
06 END IF

2.1.4 Proof of Correctness for Qiu’s Merge Procedure

A – Input array;
a1 – Auxiliary array;
s – Starting index of A;
m – Middle index of A;
e – Ending index of A;
i – Index of next available slot in workspace;
j – Index of the next element in right subarray;
k – Index of the next element in a1

THEOREM 2.1. Array A is sorted if the first while loop terminates at case 1 where k equals the size of a1

Since \( k \) equals to \( a1.size() \), each element in \( a1 \) must have been compared with some elements in the right subarray, then copy back to \( A \) because elements in \( a1 \) are smaller. Thus, we have \( a1[k-1] \leq A[j] \). In conclusion, \( A[s] \leq A[s+1] \leq ... \leq A[i-1] \leq A[j] \leq ... \leq A[e] \), which implies the array is sorted.

**THEOREM 2.2. If the first while loop terminates at case 2 where \( j \) equals \( e+1 \), then there remains element(s) in \( a1 \) that hasn’t been copy back to \( A \) and we can finish sorting \( A \) by placing remaining element(s) in \( a1 \) back to \( A \) in the way suggested by pseudocode above.**

**PROOF.** When loop terminates, we have \( s < i < e \) and \( A[s] \leq A[s+1] \leq ... \leq A[i] \). According to the algorithm, each element in the right subarray had been compared with some elements in \( a1 \), then elements in the right subarray were moved to other locations in \( A \) because they are smaller. Thus, there must exists at least one element in \( a1 \) that hasn’t been copy back to \( A \) when while loop terminates. Because \( a1 \) is the exact copy of the left subarray, and by the nature of the merge sort, left and right subarrays are already sorted before getting into the merge procedure. Thus, we have \( 0 \leq k < a1.size() - 1 \) and \( a1[k] \leq a1[k+1] \leq ... \leq a1[a1.size() - 1] \). \( A[i-1] \) must be less than \( a1[k] \) because otherwise, \( a1[k] \) will be copy back into \( A \). Therefore we have \( A[s] \leq A[s+1] \leq ... \leq A[i-1] \leq a1[k] \leq a1[k+1] \leq ... \leq a1[a1.size() - 1] \) and we can finish sorting \( A \) by placing remaining element(s) in \( a1 \) back to \( A \).

### 2.2 Pictorial Representation for Qiu’s Merge Procedure

In the following two figures, an example of a subarray of size eight is presented for Qiu’s merge procedure which is discussed in section 2.1.2. Figure 1 is the pictorial representation of the proposed implementation where case 1 terminates the while loop. Figure 2 is the pictorial representation of the proposed implementation where case 2 is demonstrated.

---

**Figure 1: Pictorial Representation of Qiu’s - case 1**

**Figure 2: Pictorial Representation of Qiu’s - case 2**
3 Analysis of the Implementation of Qiu’s Merge Procedure

In this section, a detailed analysis of Qiu’s implementation is covered. It starts with the pictorial representation of recursion tree, followed by the proof of theorem 3.1. It then follows by five subsections. The first subsection presents the analysis of the number of assignment operation for Qiu’s merge. The second subsection gives the analysis of the number of comparison. The third subsection covers time complexity. The fourth subsection shows the analysis of space complexity and the last subsection provides the execution time comparison.

Source code for the experimental data was written in C++11 and tested on windows 8.1, with Intel Core i5 processor and 8GB of RAM. Each input array consists numbers randomly generated using C++ random number engine with uniform integer distribution. Numbers in each input array range from 0 to the size of the array. The number of assignment operation and comparison was recorded using counters.

For the sake of simplicity, we assume the size of input array is a power of 2. Let the size of input array be \( n \), where \( n = 2^i \). As Figure. 3 indicates, the recursion tree of merge sort has \( i+1 \) levels where \( i \) equals to \( \log_2(n) \).

![Recursion tree of merge sort](image)

For \( 1 \leq j \leq i+1 \), there are total of \( 2^{j-1} \) node(s) at jth level, with each node contains \( 2^{i-j} \) elements from input array.

**Theorem 3.1** Given a recursion tree of merge sort with input size of \( n \) where \( n = 2^i \). For \( 1 \leq j \leq i+1 \), there are total of \( 2^{j-1} \) node(s) at jth level, with each node contains \( 2^{i-j} \) elements from input array.

**PROOF.** Base case: When \( j = 1 \), there is \( 2^{j-1} = 1 \) node at level 1 (root) which contains \( 2^{i-1} = 2^i \) elements. Induction step: Let \( j = k \) where \( 1 < k \leq i \) and assume that there are total of \( 2^{k-1} \) nodes at kth level, with each node contains \( 2^{i-k} \) elements from input array. Consider how dividing procedure of merge sort works, the number of nodes at level \( k+1 \) is \( 2 \) times as many as the number of nodes at level \( k \), which equals to \( 2^{k-1} \). Similarly, the number of elements in each node at \( k+1 \) level is half the size of that at \( k \)th level, which equals to \( 2^{i-k} / 2 = 2^{i-k-1} \). By the principle of induction, for \( 1 \leq j \leq i+1 \), there are total of \( 2^{j-1} \) node(s) at jth level, with each node contains \( 2^{i-j} \) elements from input array.

3.1 Analysis of the Number of Assignment Operations

3.1.1 Conventional Cormen’s Merge

To merge each pair of subarrays where \( n_i \) is sum of the size of two subarrays, it takes \( n_i \) assignments to copy each element to auxiliary arrays, and \( n_i \) assignments to copy each element in auxiliary arrays back to input array. Thus, the total number of assignments of one invocation of the merge procedure is \( 2 * n_i \).

Given an input array with size \( n \), where \( n = 2^i \), the recursion tree generated has \( i + 1 \) levels. For \( 1 \leq j \leq i+1 \), there are total of \( 2^{j-1} \) node(s) at jth level, with each node contains \( 2^{i-j} \) elements from input array. Starts from the second lowest level (the lowest level contains leaves), the number of assignments at jth level where \( 1 \leq j < i+1 \) is \( 2^{j-1} * 2 * 2^{i-j} = 2^{i+1} \). Thus, the total number of assignment operations of merge sort equals to \( i * 2^{i+1} \). We have \( i = \log_2(n) \) and \( 2^i = n \) which implies the total number of assignments equals \( 2 * \log_2(n) * n \).

3.1.2 The Proposed Qiu’s Implementation

To merge each pair of subarrays which \( n_i \) is sum of the size of two subarrays, it takes \( \frac{n_i}{2} \) assignments to copy each element in the left subarray to auxiliary arrays, and at most \( n_i \) assignments (\( \frac{n_i}{2} \) in the best case) to move elements in the auxiliary array and right subarray to the right position in input array. Thus, in the worst case, the total number of assignments of one invocation of the merge procedure is \( \frac{3}{2} * n_i \).

Given an input array with size \( n \), where \( n = 2^i \), the recursion tree generated has \( i + 1 \) levels. For \( 1 \leq j \leq i+1 \), there are total of \( 2^{j-1} \) node(s) at jth level, with each node contains \( 2^{i-j} \) elements from input array. Starts from the second lowest level, in the worst case, the total number of assignments at jth level where \( 1 \leq j < i+1 \) is \( 2^{i-j} * 2^{i-j} = 3 * 2^{i+1} \). Thus, in the worst case, the total number of assignments of proposed algorithm equals to \( i * 3 * 2^{i-j} = \frac{3}{2} * 2^i \). We have \( i = \log_2(n) \) and \( 2^i = n \) which implies the total number of assignments equals \( \frac{3}{2} * \log_2(n) * n \).
Experiments were run on data size between 1,000,000 to 7,000,000 for both Corman’s merge and Qiu’s merge, and the number of assignment operations are shown in Table. 1

Table 1: Number of Assignment Operations of Corman’s Merge vs. Qiu’s Merge

<table>
<thead>
<tr>
<th>Input size</th>
<th>Corman’s</th>
<th>Qiu’s</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000,000</td>
<td>39902848</td>
<td>29412692</td>
<td>10490156</td>
</tr>
<tr>
<td>2,000,000</td>
<td>83805696</td>
<td>61824950</td>
<td>21980746</td>
</tr>
<tr>
<td>3,000,000</td>
<td>129611392</td>
<td>96266058</td>
<td>33345334</td>
</tr>
<tr>
<td>4,000,000</td>
<td>175611392</td>
<td>129648199</td>
<td>45963193</td>
</tr>
<tr>
<td>5,000,000</td>
<td>223222784</td>
<td>165759201</td>
<td>57463583</td>
</tr>
<tr>
<td>6,000,000</td>
<td>271222784</td>
<td>201526805</td>
<td>69695979</td>
</tr>
<tr>
<td>7,000,000</td>
<td>319222784</td>
<td>236889067</td>
<td>82333717</td>
</tr>
</tbody>
</table>

3.2 Analysis of the Number of Comparison

The analysis is performed based on pseudocode of conventional Merge Sort and Proposed Algorithm presented in section 2. Only comparisons between elements in auxiliary arrays and input array was counted.

3.2.1 Conventional Corman’s Merge

The number of comparison performed in merging each pair of subarrays is \( n_i \), where \( n_i \) is the sum of the size of two subarrays.

Based on the theorem 2.1, for \( 1 \le j \le i+1 \), there are total of \( 2^{i-j} \) node(s) at jth level, with each node contains \( 2^{i-j} \) elements from input array. Starts from the second lowest level, the total number of comparisons at jth level where \( 1 \le j < i+1 \) is \( 2^{i-j} * 2^{i-j} = 2^i \). Thus, the total number of assignments equals to \( i * 2^i \). We have \( i = \log_2(n) \) and \( 2^i = n \) which implies the total number of assignments equals \( \log_2(n) * n \).

3.2.2 Proposed Qiu’s Merge

The number of comparison performed in merging each pair of subarrays is at most \( n_i \), where \( n_i \) is the sum of the size of two subarrays.

Based on the theorem 2.1, for \( 1 \le j \le i+1 \), there are total of \( 2^{i-j} \) node(s) at jth level, with each node contains \( 2^{i-j} \) elements from input array. Starts from the second lowest level, in the worst case, the number of comparisons at jth level where \( 1 \le j < i+1 \) is \( 2^{i-j} * 2^{i-j} = 2^i \). Thus, the total number of comparison equals to \( i * 2^i = O(\log_2(n) * n) \).

Table 2 show the number of comparison of conventional merge sort and proposed algorithm. We chose input size as power of 2 for first five test solely for mathematical purpose because it is easier to check the correctness of the data.

Table 2: Number of Comparison of Corman’s vs. Qiu’s

<table>
<thead>
<tr>
<th>Input Size</th>
<th>Corman’s</th>
<th>Qiu’s</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>2048</td>
<td>1726</td>
<td>322</td>
</tr>
<tr>
<td>512</td>
<td>4608</td>
<td>3948</td>
<td>660</td>
</tr>
<tr>
<td>1024</td>
<td>10240</td>
<td>8947</td>
<td>1293</td>
</tr>
<tr>
<td>2048</td>
<td>22528</td>
<td>19971</td>
<td>2557</td>
</tr>
<tr>
<td>10,000</td>
<td>133616</td>
<td>120450</td>
<td>13166</td>
</tr>
<tr>
<td>100,000</td>
<td>1668928</td>
<td>1535855</td>
<td>133073</td>
</tr>
<tr>
<td>1,000,000</td>
<td>19951424</td>
<td>18675192</td>
<td>1276232</td>
</tr>
<tr>
<td>5,000,000</td>
<td>111611392</td>
<td>105051895</td>
<td>6559497</td>
</tr>
</tbody>
</table>

3.3 Analysis of Time Complexity

Let \( T(n) \) be the running time of performing merge sort on an array of size \( n \). We have \( T(1) = 1 \), because it takes constant time to sort one element. According to subsection 3.1 and 3.2, on an array of size \( n \), the number of assignment operations of merge procedure is \( \frac{3n}{2} \) in the worst case. The number of comparisons operations of merge procedure is \( n \) in the worst case, thus the running time of merge procedure at each level is \( \theta(n) \). We have:

\[
T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
2 \cdot T \left( \frac{n}{2} \right) + \theta(n) & \text{if } n > 1 
\end{cases}
\]

By applying the master theorem, we get

\[
T(n) = n \cdot \log_2(n).
\]

3.4 Analysis of Space Complexity

Suppose each element in auxiliary arrays takes \( c \) unit space where \( c \) is a constant. The total extra space required to merge a pair of subarrays is \( c \cdot \frac{n_i}{2} \) unit space, where \( n_i \) is the sum of the size of two subarrays.

Given an input array with size \( n \), where \( n = 2^i \), the recursion tree generated has \( i + 1 \) levels. For \( 1 \le j \le i+1 \), there are total of \( 2^{i-j} \) node(s) at jth level, with each node contains \( 2^{i-j} \) elements from input array. Because the auxiliary is destroyed after each invocation of merge procedure, starts from the second lowest level, the space required at jth level where \( 1 \le j < i+1 \) is \( c \cdot \frac{1}{2} \cdot 2^{i-j} \). When \( j=1 \), the space complexity is \( c \cdot \frac{1}{2} \cdot 2^i = c \cdot \frac{n}{2} = O(n) \).

3.5 Execution Time Comparison

Table. 3 presents the execution time in seconds of Merge Sort using Corman’s merge and Qiu’s merge for data size between 1,000 to 15,000,000. Figure 4 shows the corresponding comparison in graph.
Table 3: Execution time in seconds of Merge Sort using Cormran’s vs. Qiu’s

<table>
<thead>
<tr>
<th>Input Size</th>
<th>Merge Sort</th>
<th>Proposed Algorithm</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>0.005003</td>
<td>0.002022</td>
<td>0.002981</td>
</tr>
<tr>
<td>10,000</td>
<td>0.027018</td>
<td>0.013028</td>
<td>0.01399</td>
</tr>
<tr>
<td>100,000</td>
<td>0.269208</td>
<td>0.139109</td>
<td>0.130099</td>
</tr>
<tr>
<td>1,000,000</td>
<td>2.86405</td>
<td>1.44703</td>
<td>1.41702</td>
</tr>
<tr>
<td>5,000,000</td>
<td>16.3936</td>
<td>8.91332</td>
<td>7.48028</td>
</tr>
<tr>
<td>8,000,000</td>
<td>27.0602</td>
<td>13.6107</td>
<td>13.4495</td>
</tr>
<tr>
<td>11,000,000</td>
<td>37.4896</td>
<td>21.6163</td>
<td>15.8733</td>
</tr>
<tr>
<td>13,000,000</td>
<td>42.5572</td>
<td>23.7749</td>
<td>18.7823</td>
</tr>
<tr>
<td>15,000,000</td>
<td>54.5087</td>
<td>30.4256</td>
<td>24.0831</td>
</tr>
</tbody>
</table>

Figure 4: Execution Time Comparison of Merge Sort between Cormran’s vs Qiu’s

From Table 3 and Figure 4, it can be found that the proposed Qiu’s implementation is faster than the conventional Cormran’s implementation and Qiu’s implementation can reduce the execution time by up to 44.18% when the input data size of randomly generated integers is 15,000,000.

4 Conclusion

This paper proposes an enhanced implementation for conventional merge sort algorithm. Even though the overall algorithm time complexity and space complexity are not changed, both the analysis and the experimental data results show the proposed implementation successfully sorts elements in ascending order with lower execution time than the conventional implementation due to the reduction of the number of auxiliary array, fewer assignment operations and fewer comparisons. Our future work will include the comparison of our proposed merge sort implementation with quick sort to examine its performance. We also plan to extend our work into parallel execution to further investigate its efficiency.

Acknowledgement

We would like to acknowledge all our paper reviewers for their constructive feedbacks.

Reference