2003 International Symposium on Advanced Intelligent Systems
September 25-28, 2003, Jeju, Korea

A New Method for Efficient in-Place Merging

Pok-Son Kim, Arne Kutzner
Kookmin University, Department of Mathematics, Seoul 136-702, Korea
pskim@kookmin.ac.kr
Seokyeong University, Department of E-Business, Seoul 136-704, Korea
kutzner@skuniv.ac.kr

Abstract

There is a well-known simple, stable standard merge algorithm, which uses only linear time but for the price of double space. This extra space consumption has been often remarked as lack of the standard merge sort algorithm that covers a merge process as central operation. In-place merging is a way to overcome this lack and so is a topic with a long tradition of inspection in the area of theoretical computer science.
We present an in-place merging algorithm that rearranges the elements to be merged by rotation, a special form of block interchanging. Our algorithm is novel, due to its technique of determination of the rotation-areas. Further it has a short and transparent definition.
We will give a presentation of our algorithm and prove that it needs in the worst case not more than twice as much comparisons as the standard merge algorithm. Experimental work has shown that our algorithm is efficient and so might be of high practical interest.

Introduction

The well-known standard merge algorithm is of linear time but needs linear extra space. This need of extra space is often remarked as significant lack of the standard algorithm and motivated the search for efficient in-place merging algorithms. A brief overview of the historical developments in the search for such algorithms gives the introduction of [3]. Roughly spoken an algorithm merges *in-place* if it needs only a constant amount of extra dataspace. Additionally the stack is restricted to logarithmic depth.
Most, but not all (e.g. [5]), in-place merging algorithms proposed so far are block-interchange based. Basic idea of all these algorithms is to realize the merge process as a sequence of successive different block interchanges. Special forms of block interchanges are rotations. A rotation ensures that the ordering of the elements doesn't change during block interchange. A description of several rotation algorithms is contained in [2].
We will introduce a rotation-based in-place merge algorithm. The algorithm uses a novel technique for the determination of the areas to be rotated. The novel technique bases on a strategy of symmetrical comparisons. Our algorithm has a short and transparent specification in contrast to the algorithms presented in [3], [6] and [7]. Experimental work has shown that it is efficient and fast.
A merging algorithm is regarded as *stable*, if it preserves the initial ordering of elements with equal keys. We assume that our algorithm is stable, but won't give any proof for this assumption here.
First we will describe our algorithm and the basic principle of its block determination. Then, giving a sketchy proof, we will show that our algorithm is of linear time in the worst case. Finally we will report about some experimental work.

Algorithm

The presented algorithm will be called *Powermerge*, due to its efficiency compared to most other merge algorithms proposed in the literature so far. Powermerge uses a classical divide-and-conquer strategy [4] for merging and acts according to the following principle:
In a first step special bounds inside either sequence to be merged are calculated. The second step is a interchange of elements between either sequence, realized by a rotation. The area to be rotated is determined by the bounds calculated in the first step. After the rotation the algorithm is applied recursive to smaller subsequences.

We now describe the method more in detail by using a accompanying example:
We assume that we have to merge two sequences $U = (0, 2, 5, 9)$ and $V = (1, 4, 7, 8)$. Figure 1 a) shows the picture at the beginning of the merge. When we compare the input with the sorted result, we can see that in the output the last two elements of U (the elements 5 and 9) occur on positions that belong to V, while the first two elements of V appear on positions in U. So 2 elements were moved from U to V and conversely. Figure 1 a) shows this fact graphically. The kernel idea of Powermerge is to compute the number of moving elements efficiently and to apply a rotation that exactly interchanges the corresponding elements. After the rotation the arising subsequences are merged by recursive applications of Powermerge. The rotation process as well as the recursive applications is shown by figure 1 c).

It is an important observation that we can exactly specify

Algorithm 1 POWERMERGE algorithm

```
POWERMERGE (A, from, pivot, to)
  middle ← (from + to)/2;  mp ← middle + pivot
  if pivot > middle then
     lstart ← mp - to;  lend ← middle
     rstart ← pivot;  rend ← to
     rfrom ← BOUND (A, lstart, lend, mp - 1)
     rto ← mp - rfrom
     if rstart < rto then
        ROTATE (A, rfrom, rstart, rto)
        if rfrom > from then
           POWERMERGE (A, from, rfrom, lend)
           if rto < rend then
              POWERMERGE (A, lend, rto, rend)
     else if lend < rstart then
        POWERMERGE (A, lend, rstart, rend)
  else
     lstart ← from;  lend ← pivot
     rstart ← middle;  rend ← mp - from
     rfrom ← BOUND (A, lstart, lend, mp - 1)
     rto ← mp - rfrom
     if rfrom < lend then
        ROTATE (A, rfrom, lend, rto)
        if rto < to then
           POWERMERGE (A, rstart, rto, to)
           if rfrom > lstart then
              POWERMERGE (A, lstart, rfrom, rstart)
     else if lend < rstart then
        POWERMERGE (A, lstart, lend, rstart)

BOUND (A, l, r, p)
  while l < r
     m ← (l + r) / 2
     if A[m] ≤A[p - m]
       then l ← m + 1;
       else r ← m;
  return l
```

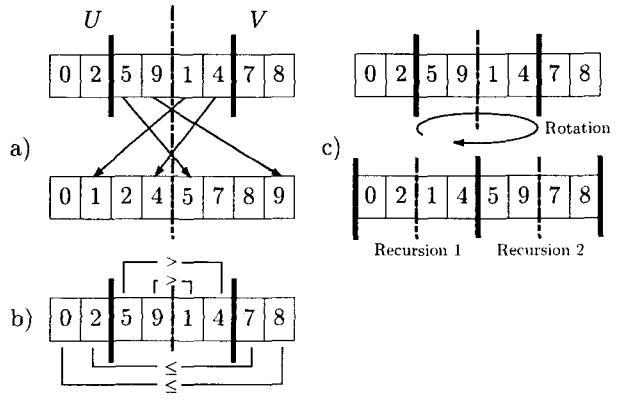

Figure 1: Powermerge example

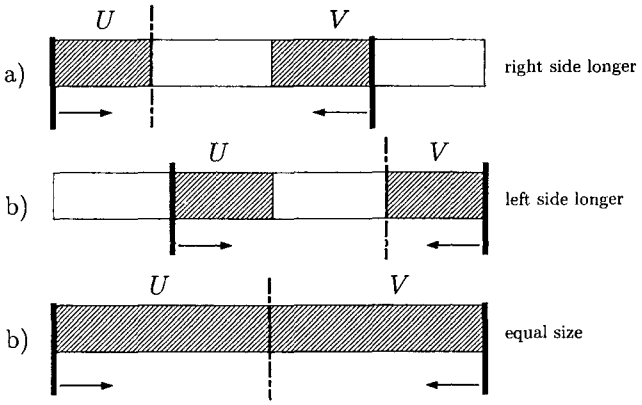

Figure 2: Cutting rules

which elements must exchanged in order to get a sorted result. If we merge two ascending sorted sequences U, V, the n greatest elements of U always move to V and the n smallest elements of V move to U, for some $n \geq 0$. This mechanism of mutual exchange is central in the context of the correctness of Powermerge.

We will now focus on the process of determining the number of elements to be exchanged. This number may be determined by a process of symmetrical comparison of elements of both sequences and happens according to the following principle:
We start at the leftmost element in U and at the rightmost element in V and compare the elements at both positions. We continue doing so by symmetrically comparing element-pair by element-pair from the outsides to the middle. Figure 1 b) shows the resulting pattern of mutual comparisons of elements in U and V. There can occur at most one position, where the relation between the compared elements alters from 'less than' to 'greater'. In our example two thick lines mark this position. This thick lines mark the bounds for the rotation. The elements between the two bounds must be interchanged to get a sorted result.
If the bounds are on the outermost left and right position, this means all elements of U are greater than all elements of V, both blocks must be exchanged and we have immediately a sorted result. Conversely, if both bounds meet in the middle we have to do nothing, because UV is then already sorted.

The idea presented so far used a sequential strategy for computing the rotation-bounds. Alternatively this can be performed more efficiently by a strategy similar to a binary search. Such a strategy is used by the function BOUND whose definition is given as part of the Powermerge-definition (see Algorithm 1). The function BOUND computes the requested bounds efficiently using $\lfloor \log_2(\min(|U|, |V|)) \rfloor + 1$ comparisons.

If the sequences to be merged are of different size, we use mechanism of cutting out a subsection of the longer input sequence. Figure2 explains this mechanism graph-

ically. This means, we handle asymmetry by operating only on a subsection of the longer sequence. In the case of a rotation, the rotation is extended to the elements between the marked areas of figure 2 .

Algorithm 1 gives a specification of the Powermerge-algorithm in pseudo code.

The Powermerge algorithm is rotation based. The rotation corresponds to a block interchange but both notions are not identical because rotation implies that the ordering of the elements is preserved. Several efficient rotation algorithms are presented in [2].

Worst Case Complexity

There have been proposed in-place merging algorithms that are not of linear time in the worst case, for example the shuffle based algorithm proposed by Ellis and Markov in [5]. We will now give a sketchy proof that Powermerge is a linear time algorithm with respect to the number of comparisons.

Theorem 1: *Let $n = 2m, m = 2^k$. If two sequences of size m are merged, the number of comparisons needed in the worst case is $2n - k - 3 = O(n)$.*

Proof: The number of comparisons needed in the worst case is: $(\log m+1)+2((\log(m/2))+1)+2^2((\log(m/2^2))+1)+\cdots+2^k((\log(m/2^k))+1)$ where $m = 2^k$

The above expression is resolved as follows:

$(\log m + 1) + 2((\log(m/2)) + 1) + 2^2((\log(m/2^2)) + 1) + \cdots + 2^k((\log(m/2^k)) + 1)$

$= \sum_{i=0}^{k}(2^i(\log 2^{k-i} + 1))$

$= \sum_{i=0}^{k} 2^i(\log 2^{k-i}) + \sum_{i=0}^{k} 2^i$

$= \sum_{i=0}^{k} 2^i(k - i) + \sum_{i=0}^{k} 2^i$

$= k * \sum_{i=0}^{k} 2^i - \sum_{i=0}^{k} i * 2^i + \sum_{i=0}^{k} 2^i$

$= (k + 1)(\frac{2^{k+1}-1}{2-1}) - ((k - 1)2^{k+1} + 2)$

$= (k + 1)2^{k+1} - (k + 1) - (k - 1)2^{k+1} - 2$

$= 2 * 2^{k+1} - k - 3 = 2n - k - 3 = O(n)$

qed.

Experimental results

First experimental work with the Powermerge algorithm has shown, that this algorithm might be of high practical interest. Possibly it is the fastest in-place merge algorithm known so far. We compared our algorithm to the rotation-based algorithm contained in the C++ STL-library [1] provided by SGI, which has a rather short definition and is known as fast and efficient. The Powermerge algorithm performed generally less comparisons than his STL-counterpart and consumed roughly 20% less computing time.

Further we compared a Mergesort on basis of Powermerge with Quicksort, Heapsort and other sorting algorithms. We observed that Mergesort on basis of Powermerge always needed fewer comparisons than Quicksort and Quickersort. In any situation where the input data had some presorted nature, our variant of Mergesort was the clear winner compared to all other sorting algorithms. The reason for this behavior is the property of Powermerge to detect any kind of presorting efficiently.

Conclusion

We presented an efficient in-place merging algorithm called Powermerge. Our algorithm has a much shorter definition and is less complex than the algorithms presented in [3], [6], [7]. We could prove that our algorithm is of linear time regarding the number of comparisons. During practical experimentation our algorithm has shown a very good performance.

In the future, we plan to investigate the number of element-moves performed by Powermerge. Further we plan to analyze the average number of comparisons needed by Powermerge. During our experimental work we could observe that a Mergesort basing on Powermerge always needed fewer comparisons than Quicksort. We plan to give a formal proof for that.

References

[1] http://www.sgi.com/tech/stl.

[2] J. Bentley. *Programming Pearls.* Addison-Wesley, Inc, 2nd edition, 2000.

[3] J. Chen. Optimizing stable in-place merging. *Theoretical Computer Science*, 302(1/3):191–210, 2003.

[4] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, 2nd edition, 2001.

[5] John Ellis and Minko Markov. In situ, stable merging by way of the perfect shuffle. *The Computer Journal*, 43(1):40–53, 2000.

[6] V. Geffert, J. Katajainen, and T. Pasanen. Asymptotically efficient in-place merging. *Theoretical Computer Science*, 237(1/2):159–181, 2000.

[7] J. Katajainen, T. Pasanen, and J. Teuhola. Practical in-place mergesort. *Nordic Journal of Computing*, 3:27–40, 1 1996.